Saarland University
Faculty 6 – Natural Sciences and Technology 1
Department 6.2 – Computer Science
Programming Systems Lab

Subtype Satisfiability and Entailment

Tim Priesnitz

Saarbrücken, 2004

Doctoral thesis
submitted for the Degree
Doktor der Naturwissenschaften (Dr. rer. nat.)
at the Faculty 6 – Natural Sciences and Technology 1
of the Saarland University

Directed by Privatdozent Dr. Joachim Niehren

This thesis was set in Times by the author using the LaTeX typesetting system.

An electronic version is available from `www.ps.uni-sb.de/Papers`.

This thesis was supported by the graduate studies program "Quality Guarantees for Computer Systems" at the Department of Computer Science at the University of the Saarland funded by the German Science Foundation.

Subtype Satisfiability and Entailment, Tim Priesnitz, doctoral thesis,
Saarland University, Saarbrücken, 2004.

Date of the colloquium: 1.4.2005

Dean of Faculty 6 – Natural Sciences and Technology 1

Prof. Dr. Jörg Eschmeier

Members of the committee

Vorsitzender:	Prof. Dr.-Ing. Jörg Siekmann
Erstgutachter:	Privatdozent Dr. Joachim Niehren
Zweitgutachter:	Prof. Dr. Gert Smolka
akademischer Beisitzer:	Hochschuldozent Dr. Christoph Benzmüller

---

Hiermit erkläre ich, Tim Priesnitz, an Eides statt, dass ich die vorliegende Dissertation selbstständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, December 2004

Abstract

Subtype constraints were introduced in advanced programming language research for designing subtype systems and program analysis algorithms. Two logical problems arise in this context: subtype satisfiability and subtype entailment. Subtype satisfiability underlies subtype inference; subtype entailment is for simplifying subtyping constraints in the same application.

In this thesis, we investigate both problems systematically for a number of dialects of subtyping constraint languages that may vary in the following dimensions: types may be simple (finite) or recursive (infinite), type constants may be ordered in lattices or in general partially ordered sets, subtyping can be structural or non-structural, depending on whether least and greatest types are permitted. We use and develop new formal reasoning techniques based on automata, unification, and modal logic.

Subtype satisfiability is well understood for all dialects with constants ordered in a lattice. Although cubic time algorithms are given by Palsberg and O'Keefe (1995), Pottier (1996), and Palsberg, Wand, and O'Keefe (1997), little is known about dialects where constants belong to arbitrary partially ordered sets. We present a uniform treatment to determine the complexities of all these classes. As a consequence, we settle a problem left open by Tiuryn and Wand in 1993 and also subsume complexity bounds given by Wand and Tiuryn (1993), Tiuryn (1992), and Frey (2002). Our results are based on a new connection between modal logic and subtype constraints that we present.

Subtype entailment is known to be hard even for simple subtype constraint languages. Rehof and Henglein determined the complexity of structural subtype entailment with type constants ordered in a lattice. They proved coNP-completeness for simple types (1997) and PSPACE-completeness for recursive types (1998). Furthermore, they showed that non-structural subtype entailment is PSPACE-hard and is conjectured PSPACE-complete for the case with only two type constants for the least and greatest types respectively (1998). Yet the problem still remains open today. We argue that the difficulty occurs due to effects linked to non-regular word languages. In order to do so, we precisely characterize subtype entailment by finite word automata with word equations. This characterization induces new results on non-structural subtype entailment, constituting a promising starting point for future investigation on decidability.

Ausführliche Zusammenfassung

Teiltyp-Constraints stammen aus der programmiersprachlichen Typinferenz und Programmanalyse. Hier sind zwei logische Probleme von besonderem Interesse: Erfüllbarkeit und Subsumption. Das Lösen von Teiltyp-Constraints ist eine Kernaufgabe von Typinferenzsystemen mit Teiltypen. Erfüllbarkeit wird getestet, um Programme mit Typfehlern auszuschließen. Subsumption nützt in dieser Anwendung zur Vereinfachung von Typconstraints.

Diese Arbeit untersucht beide Probleme systematisch für verschiedene Teiltyp-Constraintsprachen. Hierbei variieren wir die Typsprachen und die Teiltyp-Ordnung: Typen sind entweder einfach (endlich) oder rekursiv (unendlich); die Ordnung auf Typkonstanten ist entweder durch einen Verband festgelegt oder durch eine allgemeinere partielle Ordnung gegeben; die Teiltyp-Ordnung ist entweder strukturell oder nicht-strukturell, in Abhängigkeit davon, ob wir die Existenz des kleinsten beziehungsweise größten Typen annehmen. Wir benutzen und entwickeln spezielle Methoden der computationalen Logik, insbesondere Automaten Theorie, Unifikation, und Modallogik.

Teiltyp-Erfüllbarkeit ist für Constraintsprachen wohl verstanden, deren Typkonstanten in einem Verband angeordnet sind. Palsberg und O'Keefe (1995), Pottier (1996) und Palsberg, Wand und O'Keefe (1997) entwickelten kubische Algorithmen für derartige Probleme. Hingegen ist wenig über allgemeinere Dialekte bekannt, in denen Typkonstanten lediglich partiell geordnet sind. Wir entwickeln einen neuen universellen Ansatz, der es uns ermöglicht, die Komplexität von Erfüllbarkeit von Teiltyp-Constraintsprachen in allen genannten Dimensionen zu bestimmen.

Wir lösen ein Komplexitätsproblem zu Teiltyp-Erfüllbarkeit, das Wand und Tiuryn 1993 offen ließen. Unser Resultat beruht auf einem neuen Zusammenhang zwischen Teiltyp-Constraints und Modallogik, den wir aufzeigen. Den regulären Fall lösen wir durch äquivalente Übersetzung in ein passendes Fragment der aussagenlogischen dynamischen Logik (PDL). Erfüllbarkeit dieser PDL-Variante ist DEXPTIME-vollständig, wie wir mithilfe von Baumautomaten zeigen. Den endlichen Fall lösen wir analog durch äquivalente Übersetzung in eine Teilsprache von K-normaler Modallogik. Erfüllbarkeit dieser Modallogik beweisen wir als PSPACE-vollständig, indem wir Ideen von Spaan (1993, 2000) verallgemeinern.

Teiltyp-Subsumption ist selbst für einfache Constraintsprachen wegen seiner hohen Komplexität bekannt. Rehof und Henglein bestimmten die Komplexität für den strukturellen Verbandfall mit zwei Typkonstanten. Sie zeigten coNP-Vollständigkeit für einfache Typen (1997) und PSPACE-Vollständigkeit für rekursive Typen (1998). Desweiteren zeigten sie PSPACE-Härte im nicht-strukturellen Fall und vermuteten PSPACE-Vollständigkeit für den nicht-strukturellen Fall (ohne weitere Typkonstante). Bis heute

konnte jedoch noch nicht einmal die Entscheidbarkeit dieses Problems gezeigt werden.

In dieser Arbeit untersuchen wir nicht-strukutelle Teiltyp-Subsumption im einfachsten Fall (mit einem einzigen Typkonstruktor und ohne weitere Typkonstanten). Wir argumentieren, dass spezielle Wortgleichungen das Problem schwierig machen, die bekannterweise nicht-reguläre Phänomene verursachen. Hierzu charakterisiseren wir Teiltyp-Subsumption mittels endlicher Automaten und Wortgleichungen; unsere Charakterisierung läßt sich im $\forall\exists^*$ Fragment der erststufigen Theorie von Wortgleichungen mit regulären Constraints formulieren, welches Durnev (1995) als unentscheidbar zeigte. Alternativ können wir sie mit speziellen nicht-regulären endlichen Automaten darstellen, die wir entwickeln. Für eine Teilklasse dieser Automaten formulieren wir ein Entscheidungsverfahren für Sprachuniversalität. Wir isolieren somit ein Fragment einer nicht-strukturellen Teiltyp-Constraintsprache, für das Subsumption entscheidbar ist. Es bleibt jedoch offen, ob sich dieses Verfahren auf volle Teiltyp-Subsumption erweitern läßt. In jeden Fall liefert wir dazu einen neuen Ansatz.

Kurze Zusammenfassung

Diese Arbeit untersucht zwei logische Probleme der programmiersprachlichen Typinferenz: Erfüllbarkeit und Subsumption von Teiltyp-Constraints. Wir untersuchen diese Probleme systematisch für eine Reihe von Constraintsprachen. Dabei greifen wir auf Methoden der computationalen Logik, Unifikations- und Automatentheorie zurück.

Teiltyp-Erfüllbarkeit ist für den Fall wohl verstanden, dass die Typkonstanten in einem Verband angeordnet sind (Palsberg und O'Keefe (1995), Pottier (1996), Palsberg, Wand und O'Keefe (1997)). Der allgemeinere Fall mit beliebig angeordneten Konstanten wurde bislang weniger untersucht. Wir stellen einen ersten universellen Ansatz vor, indem wir erstmals einen Zusammenhang zwischen Teiltyp-Constraints und Modallogik aufzeigen. Dadurch lösen wir unter Anderem ein seit 1993 offenes Komplexitätsproblem von Wand und Tiuryn.

Teiltyp-Subsumption ist selbst für einfachste Constraintsprachen von hoher Komplexität. Rehof und Henglein zeigten dies für den strukturellen Verbandsfall (mit zwei Typkonstanten 1997, 1998), ließen jedoch den nicht-strukturellen Fall offen. In dieser Arbeit betrachten wir den einfachsten nicht-strukturellen Fall. Hier zeigen wir, dass versteckte Wortgleichungen neue Schwierigkeiten verursachen. Hierzu charakterisieren wir Teiltyp-Subsumption durch spezielle endliche Automaten mit Wortgleichungen. Unsere Charakterisierung liefert partielle Entscheidbarkeitsresulte zur nichtstrukturellen Teiltyp-Subsumption und kann als Grundlage für künftige Untersuchungen dienen.

Acknowledgements

First and foremost, I would like to thank my thesis advisor, Joachim Niehren for all his invaluable and dedicated support, without which this work would never have been possible. I am very grateful to Zhendong Su and Ralf Treinen, for all those enthralling and fruitful discussions. Maarten Marx and Patrick Blackburn provided very helpful comments and ideas in the field of modal logic. Furthermore, thanks go to my second supervisor Gert Smolka and all colleagues at the programming systems lab for such a stimulating atmosphere, especially to Andreas Rossberg for all our interesting discussions in type theory as well as Thorsten Brunklaus who established a great working environment. I have to thank Klaus Schulz, Jean-Marc Talbot, Sophie Tison, Viktor Kuncak, and all the anonymous referees for all their helpful and constructive comments on my papers, and Jörg Siekmann, who entranced me with his endless enthusiasm for logic during his courses. Finally, I would like to thank Martin Müller, who introduced me to the subtype problem which I present in the second part of this thesis.

Contents

1 Introduction

Type systems[Mit96, Pie02] play a central role in modern programming languages, e.g. Caml, Java, Haskell, Standard ML, Alice, etc [WL99, GJS96, Jon03, MTH90, Prob]. Types classify the values that programs can manipulate according to operations that they permit. As opposed to dynamic typing, type systems allow us to detect programming errors statically at compile time; they can be used to specify module interfaces and to validate them. Types help reasoning about programs; they can be seen as invariants preserved by program execution. The power of type systems however, is limited by the algorithmic complexity of the services they provide, which in turn depends on the expressiveness of the underlying type language. This is why programming language designers have to carefully design programming languages together with their type systems.

Type checking and type inference are the most important services of type systems [DM82, Hin69]. Type checking requires programmers to annotate types to values; the consistency of these annotations can then be validated by the type system. In Java, for instance, a programmer has to annotate types to all variables, objects, and classes. Type inference, in contrast relieves the programmer of the burden of annotating types. Languages in the ML family (Caml, SML, Haskell, Alice) require type annotations for data types and modules; automatic type inference is supported in the core language. ML style type inference relies on first-order unification.

Subtyping is induced by a binary relation on types that orders types into a hierarchy [Mit84, Mit91, WO89, Tiu92]. All values of a type belong to all its subtypes too. The same value may thus be given by multiple types, i.e., subtyping provides a form of polymorphism. Many varieties of subtyping have been investigated. Examples are the sub-classing mechanisms in object-oriented programming languages such as C++ and Java. Another example is automatic conversion of integers into floating point numbers. Subtyping is often combined with other forms of polymorphism: parametric polymorphism in Caml, modules in ML, and templates in Java.

Types may be nested recursively; they form trees. Most typically, pair, record, and function types contain other types as subtrees. Type inference for tree-shaped types without subtyping (as in SML) amounts to first-order unification, i.e., equation solving over finite trees. Type inference in the presence of subtyping becomes more difficult. First-order unification needs to be generalized when solving subtype constraints to conjunctive logic formulas that describe types and their subtype ordering [Tiu97, Ben97, TW93, KPS94, HM95, Reh98, FM90].

In this thesis, we investigate multiple dialects of subtype constraints with respect to two logical problems: satisfiability and entailment. Satisfiability testing is subsumed by constraint solving, while entailment testing was proposed for constraint simplification during type inference.

This introduction presents a short essay on subtype inference which motivates our work in subtype constraints. We summarize our contributions and discuss related work.

1.1 Types in programming languages

Type systems [Mit96, Pie02] play a central role in modern programming languages. This applies for statically typed languages such as Standard ML, Caml, Java, Haskell, Alice, Mercury, and Scala [MTH90, WL99, GJS96, Jon03, Prob, SHC95, Proa] as well as for dynamically typed languages such as Mozart, Python and Erlang [Moz99, sf, sE].

1.1.1 Type systems

We are mainly interested in programming languages with static type inference, the majority of which belong to the ML family (Caml, Haskell, Standard ML, and Alice). The type systems of such languages is decomposed into a type system for basic values (integers, strings, tuples, functions, etc.) and an orthogonal type system for modules.

Module types specify the interface of a module, i.e., the values of a module that can now be used. Interfaces are to be defined explicitly by the programmer. The module type system provides type checking, which tests whether the implementation of the module is consistent with its interface.

The type system of the base language allows for the explicit definition of a recursive data type, and type inference for all other types, including the inference of type schemas for resolving parametric polymorphism.

For instance, we can define function composition in the core language of O'Caml, a dialect of ML, such that $\texttt{compose}(\texttt{f},\texttt{g})(\texttt{u}) = \texttt{f}(\texttt{g}(\texttt{u}))$ for all functions f, g such that outputs of f are accepted as inputs of g and values u in the input type of f:

```
val compose = fun f -> fun g -> fun u -> g(f(u))
```

This functions is polymorphic in that it can be given many types rather than a unique type. More precisely, for every type x, y, z it can be assigned to the following type:

$$\texttt{compose}: \ (x \to y) \to (y \to z) \to (x \to z)$$

We could also look at this type as a type schemata that is parametric in polymorphic type variables x, y, z from an infinite repository.

Our second example, again purely functional, shows an counter object in O'Caml. Counters are parametrized by a number n; they consist of a `get` method which returns the up-to-date counted number `n` and an `inc` method which gives access to a similar counter but incremented by 1:

```
let rec counter = fun n ->
   object
     method get = n
     method inc = counter (n+1)
   end
```

More formally, `counter` is defined as a function that creates objects, i.e. records of values that here are called methods. The argument `n` received at `counter` creation needs to be an integer. Otherwise, the term `n+1` used in the `inc` method were ill-typed. The number `1` is an integer in O'Caml, not a float as `1.0`. Addition is permitted only on numbers of the same type. The function `counter` should thus be assigned to the type $int \to \tau$ where τ is the record's type:

$$\tau = \ < \text{get} : int; \ \text{inc} : \tau >$$

Records are like tuples but at the same time they associate names to tuple components. Similarly, record types extend on tuple types. Note that records may be recursive as well as their types.

In this thesis we will mostly ignore the naming aspect of records, so that we can simplify them into tuples. Let $\times$ the constructor for pair types. We then represent the recursive record type above by the recursive pair type below:

$$\tau = int \times \tau \tag{1}$$

In contrast to good practice in O'Caml and many other programming languages, but in agreement with standard mathematical convenience, one could imagine that number `1` is polymorphic, either a integer or a float. In this case, the above `counter` would become polymorphic too.

Other forms of polymorphism are highly relevant to modern type systems, even though less important for type inference: existential types are mainly used for information hiding in module systems, dependent types [Mac86] form the foundation for the ML module type system, and row polymorphism is the foundation for subtyping objects in O'Caml [RV98].

1.1.2 Principal types in type inference

In this thesis, we focus on aspects of type inference, one of the most preferable service of type systems. In order to deal with existing polymorphism compositionally it is important to infer *most general types*, that are often called *principal types*.

The above type schema for `compose` function was principal; it is automatically inferred by O'Caml's type inference. The recursive type of `counter` objects is principle, too, but unique. It is again inferred by O'Caml's type system, which does not reject recursive types in objects.

This thesis contributes to type inference for basic values. Type quantifiers as in parametric polymorphism or module types are not in scope. As a consequence, we restrict ourselves to fairly unexpressive type languages, those that are relevant to the type inference.

First of all, there are type constants; these are types of primitive data objects like integers, characters, strings, floating point units. Second, types τ_1, τ_2 can be composed into pairs types $\tau_1 \times \tau_2$ or more general tuple types, which are able to encode record types where labels are restricted to consecutive numbers. Third, types can be composed into function types $\tau_1 \to \tau_2$, i.e., the type of functions from τ_1 to τ_2. Function types are most important in programming languages where functions are first-class values (as in the ML family), that can be passed around, computed at run time, and nested into data structures. Finally, there are recursive types, which are like the previous types but possibly of infinite nesting depth.

1.1.3 Representing types as trees

Our example of a counter shows that types may be nested recursively so that they form trees. Simple types are finitely nested; they are finite trees, i.e., ground terms build from constants, pair, tuple and function type constructors.

Recursive types can be modeled by infinite trees over the same signature. Alternatively, we can consider the class of regular trees, i.e. possibly infinite trees with at most finitely many different subtrees. Regular trees permit finite representations, by recursive equation systems or equivalently by mu terms [KPS95]. Note that we do not permit recursive types with sums (i.e., disjoint unions of types, as needed for defining the type of lists). This limitation in expressiveness is essential for feasibility of type inference.

For instance, the recursive equation (1) is satisfied by the following regular tree (left) which is finitely represented by a rooted labelled graph (right):

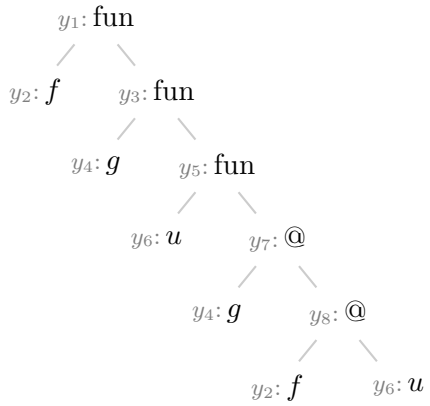
1.1.4 Type inference as constraint satisfaction

Traditionally in the lambda calculus, two styles of type systems were investigated. In Church style, all program variables are explicitly annotated by types; the consistency at these annotations is then checked by a type system. In Curry style, no types are annotated, so they need to be reconstructed from the program expression by type inference. This is the task of many contemporary type systems in todays high-level programming languages.

Type inference is usually organized in two phases. The first phase is syntax-driven: fresh type variables are assigned to all subexpressions of a program; type constraints for these variables are then inferred from the program and its annotation by type variables. In the case of the lambda calculus, for instance, one may wish to impose equality between the requested and obtained argument type of functions.

In the second phase, the accumulated set of type constraints is send to a constraint solver. The solver will solve the type constraints, or simply test their satisfiability. Programs with unsatisfiable type constraints are rejected as ill-typed. For well-typed programs, the types of values may be displayed on demand. In any case, type inference has to solve constraint satisfaction problems.

In case ML style type inference, constraint solving is restricted to first-order unification; most general unifier represent principal types. For instance, let us derive the principal type of the ML expression `compose` given above. We start to label each node in the standard tree representation of `compose` by a type variable $y_1, \ldots, y_8$. We always use fresh type variables except for leaves labeled by the same constant:



The type of `compose` needs to satisfy the schema:

$$y_1 = y_2 \to y_4 \to y_6 \to y_7.$$

The two application nodes prove that f and g are functions satisfying the following type constraints:

$$y_4 = y_8 \to y_7$$
$$y_2 = y_6 \to y_8$$

Solving all these equations yields what we expected, with somehow arbitrary names for type variables:

$$y_1 = (y_6 \to y_8) \to (y_8 \to y_7) \to y_6 \to y_7$$

Since the set of generated constraints is linear in the size of the program, the typing's complexity is chiefly determined by the complexity of constraint satisfiability. The constraints of first-order unification can be solved efficiently in quasi-linear time. A challenge these days is to design more expressive type systems that still permit type inference with low time and memory complexity.

1.2 Subtype systems and constraints

Subtyping in the context of programming languages dates back to the 1960s, where it appeared in Simula [DN66] and its relatives. Subtype systems were proposed in the early 1980's by Reynolds [Rey80] and Cardelli [Car88]. The goal was to enrich the expressiveness of type languages in type inference systems. The interest on subtyping grew in the 1990s when studying type systems for object-oriented programming languages [CM94]. Independently, subtyping received some interest for program specification [GM96] and computational linguistics [PS94].

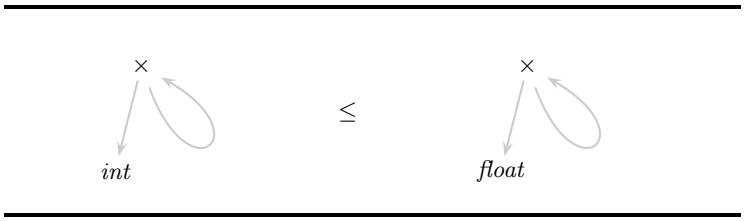


Fig. 1.2.1. An example for structural recursive subtyping.

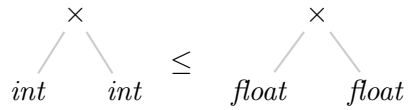
1.2.1 Subtype orderings

Types are syntactic counterparts of sets: a type represents the set of all its values. Subtyping [Mit84] is an ordering on types that corresponds to set inclusion. This means that all values of a subtype do also satisfy all supertypes. Subtyping introduces a form of polymorphism. Functions with domain type τ must be sufficiently polymorphic to accept all arguments in subtypes of τ.

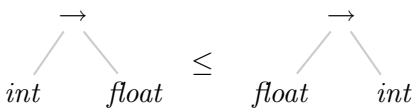
Atomic subtyping is a subtype order on type constants. In mathematics, for instance, one usually considers integers as real number. This can be expressed by the subtyping relation:

$$int \leq float$$

Structural subtyping [Mit84, AC93, Pot96, HR97] lifts atomic subtyping structurally to tuple and function types. If integers are floats then pairs of integers should be pairs of floats. This is expressed by the structural subtype relation $int \times int \leq float \times float$:



Similarly, a function of type $float \to int$ should be a function of type $int \to int$ simply by restricting its domain. In turn, such functions should be of type $int \to float$ by extending its co-domain. Note that the domains are contra-variant: smaller domains yield larger function types. Co-domains, in contrast, are co-variant: larger co-domains yield larger function types.



Two examples for structural recursive subtyping are given in Figures 1.2.1 and 1.2.2. The first use co-variant the second contra-variant function symbols. In both examples,

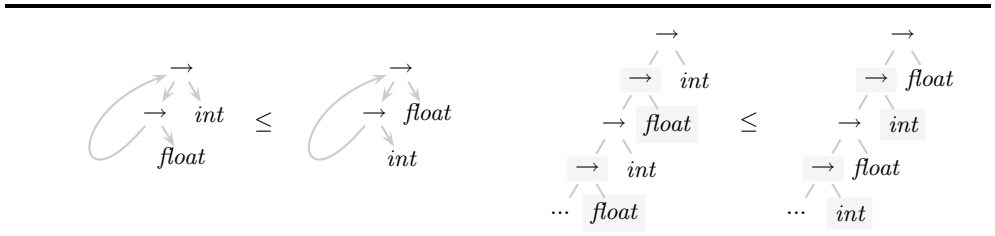


Fig. 1.2.2. Structural subtyping ordering $\leq$ on regular trees given by rooted graphs and their infinite unfoldings. All anti-monotonic nodes are grey underlined.

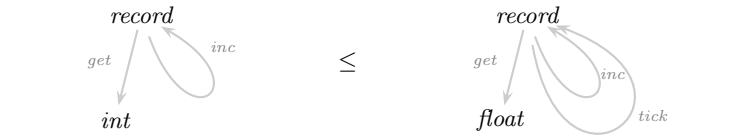


Fig. 1.2.3. Non-structural recursive subtyping on record types:
`(counter n)` yields a subtype of `(timer n)` when assuming $\mathit{int} \leq \mathit{float}$.

related types have the same structure so that integers can be recursively converted into floats.

Non-structural subtyping can lift atomic subtyping to objects, so that objects can be extended by methods while increasing their type. For illustration, we continue to assume $\mathit{int} \leq \mathit{float}$ in contrast to O'Camls practice. We can then refine counters into timers, with `inc` methods to count hours and `tick` methods to count minutes, i.e., 1/60 of hours.

```
let rec timer = fun n ->
   object
     method get  = n
     method inc  = timer (n+1)
     method tick = timer (n+1/66)
   end
```

Such timers are ill-typed in O'Caml but well-typed when assuming $\mathit{int} \leq \mathit{float}$. Thanks to subtyping, `timer` can be assigned to the type $\mathit{float} \to \tau$ where:

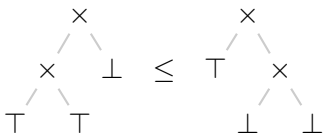
$$\tau = \ < \text{get} : \mathit{float};\ \text{inc} : \tau;\ \text{tick} : \tau >$$

The recursive type pictured on the right of Figure 1.2.3 types the application `(timer n)` for arbitrary floats `n`. We assign it to be a subtype of the type of `(counter n)` illustrated on the left of Figure 1.2.3. The subtyping relation reflects that timers have more methods than counters.

A technically simpler version of non-structural subtyping is obtained when assuming the existence of the least type $\bot$ or the greatest type $\top$ which satisfy for all other types τ:

$$\bot \leq \tau \leq \top$$

An example is the following non-structural subtype relationship that clearly orders trees with different shapes:



Many other forms of subtyping were proposed in the literature. O'Caml uses row polymorphism [RV98], the XML processing language CDuce is based on semantic subtyping [FCB02]. Partial types consist a least type but not a greatest type; they were considered in type inference [Tha94] and program flow analysis [PO95].

1.2.2 Summary

Let us summarize the subtype orderings that we will investigate in this thesis. We will study all versions of structural and non-structural subtyping presented so far, except for record types. A type τ will be a possibly infinite tree over the following vocabulary:

1. constants in a partially ordered set $(C, \leq_C)$
2. the pair type constructor $\times$,
3. the function type constructor $\rightarrow$,
4. the least and greatest type $\bot, \top$.

The subtyping relation for such types will be the largest relation $\leq$ that satisfies the following rules for all constants $c_1, c_2 \in C$ and types $\tau, \tau_1, \ldots, \tau_4$:

constants:	$c_1 \leq c_2$ iff $c_1 \leq_C c_2$
pair types:	$\tau_1 \times \tau_2 \leq \tau_3 \times \tau_4$ iff $\tau_1 \leq \tau_3$ and $\tau_2 \leq \tau_4$;
function types:	$\tau_1 \rightarrow \tau_2 \leq \tau_3 \rightarrow \tau_4$ iff $\tau_3 \leq \tau_1$ and $\tau_2 \leq \tau_4$;
least and greatest types:	$\bot \leq \tau \leq \top$.

We call subtyping *structural* if the least and greatest type are not permitted and *non-structural* otherwise. We will consider subtyping relations where the constants are ordered in lattices or else in posets. We will investigate simple types where all trees are finite, regular recursive types where all trees have at most finitely many distinct subtrees, and arbitrary recursive types.

1.2.3 Subtype constraints

Subtype constraints are positive conjunctive logical formulas that describe types via their subtype relation. The base ingredients of subtype constraint are type variables x, y, z.

Let terms t be build from variables and the constructors of our vocabulary:

$$t ::= x \mid c \mid t_1 \times t_2 \mid t_1 \rightarrow t_2 \mid \bot \mid \top$$

A subtype constraint φ is a conjunction of subtype literals:

$$t_1 \leq t'_1 \wedge \ldots \wedge t_n \leq t'_n$$

A solution of a subtype constraint φ is a variable assignment into types that renders all literals of φ true. A subtype constraint is satisfiable if it has a solution. Entailment $\varphi \models \varphi'$ holds if all solutions of φ do also solve φ'.

This thesis studies two logical problems for subtype languages: satisfiability and entailment. Satisfiability is the question of whether a given subtype constraint is satisfiable. Entailment is the question for two input constraints φ and φ' whether $\varphi \models \varphi'$.

Subtype satisfiability has been exhaustively studied for constants that are ordered in a lattice [Tiu92, PWO97, PO95, JP99]. Partial complexity results are known for the more general case of constants in posets [Tiu92, TW93, Fre02]. All missing results will be contributed in this thesis.

Henglein and Rehof [HR97, HR98] showed that structural subtype entailment is hard. In the simplest finite case they proved coNP-hardness, and in the recursive case PSPACE-completeness. Whether non-structural subtype entailment is decidable remains open until today. This was one of the motivating questions behind the research of this thesis.

1.2.4 Subtype inference

Subtype inference [Tiu97, Ben97, TW93, KPS94, HM95, Reh98, FM90, Pot98b, Su02] is considered to be practical if it can be reduced to a polynomial time subtype satisfiability problem. This imposes strong restrictions on the constant ordering in practice [KPS94, Pot96, Mit91].

The type inference algorithms of ML can be easily adapted in order to take subtyping into account. It is sufficient to consider application nodes:

Previously, we have proposed to impose the type equality $y = (z \rightarrow x)$. In the perspective of subtyping is is sufficient to require a subtype constraint $y \leq (z \rightarrow x)$.

In practice, even cubic time subtype constraint solver turned out to be problematic. Subtype simplification was proposed as a way out of the trouble [EST95a, Reh97, Pot96, Pot98a]. Simplification operations can be formulated on the basis of subtype entailment [Pot98b, NP99, NP03, HR97, HR98, Reh98, Pot96, TS96, FF99].

Rehof [Reh98] calls satisfiability to be the combinatorial bottleneck problem in deciding typability and entailment the combinatorical bottleneck problem in representing typings.

1.3 Contribution

We contribute to the open questions on subtype satisfiability and entailment. Our key results show the following:

- recursive structural subtype satisfiability over posets is DEPTIME-complete. This settles a open problem form 1993 by Tiuryn and Wand [TW93].
- Non-structural subtype entailment over the restrictive signature $\{\bot, \times, \top\}$ is equivalently characterizable by universality of regular expressions augmented by certain word equations. This characterization induces new results on an extensively attacked problem in [Su02, Reh98, HR97, HR98, Pot96, Pot98b, Pot01, TS96, FF99, AWP99, FA96, MW97].

In what follows, we will discuss both results in some more details and related them to existing results in the literature.

	structural lattice ordering	*non-structural lattice ordering*
finite trees	$O(n^3)$ Tiuryn [Tiu92]	$O(n^3)$ Palsberg, Wand, O'Keefe [PWO97]
regular or infinite trees	$O(n^3)$ Jim,Palsberg [JP99]	$O(n^3)$ Palsberg,O'Keefe [PO95]

Table 1.1: Subtype satisfiability over a lattice.

1.3.1 Subtype satisfiability

Subtype satisfiability is well understood when constants are ordered in a lattice, see Table 1.1. All four case can be solved by the methods given in [JP99].

Cubic time algorithms for non-structural subtype satisfiability are given by Palsberg, Wand, and O'Keefe [PWO97] for finite trees and by Palsberg and O'Keefe [PO95] for regular trees and one constant only. All satisfiabilty results in the lattice case are based on a closure algorithms for checking consistency. Tiuryn [Tiu97] and Benke [Ben97, Ben99] also give a polynomial algorithms for representing solution by finite automata.

For subtype satisfiability over posets, there exist only partial answers, see Table 1.2. Tiuryn and Wand show that recursive structural satisfiability is in *DEXPTIME* [TW93]. Tiuryn shows that finite structural satisfiability is *PSPACE*-hard [Tiu92], and subsequently Frey shows that it is in *PSPACE* and thus *PSPACE*-complete [Fre02]. Frey also makes use of his *PSPACE* algorithm in a type inference system for Jazz, an academic programming language for hardware verification [Fre].

Decidability and complexity of non-structural subtype satisfiability are open for both finite and recursive types.

We base our results on a new approach, connecting subtype constraints and modal logic. As an intermediate result we introduce a new subtype language of uniform subtype constraints and show that their satisfiability problem is polynomial time equivalent to that of a dialect of propositional dynamic logic [FL79, BMV94, BAHP82], which is subsumed by the monadic second-order logic SnS of the complete infinite n-ary tree [Rab69]. With this connection, we completely characterize the exact complexity of subtype satisfiability over posets in all cases: finite versus recursive types, and structural versus non-structural orderings.

Second, we reprove NP-completeness of subtype satisfiability over constants ordered over a poset where we exclude all constructors $\times$, $\rightarrow$ [PT96]. Here our characterization by modal logic scales down to Boolean logic.

	structural ordering over posets	*non-structural ordering over posets*
finite trees	in PSPACE Frey [Fre02]	PSPACE-complete [this thesis]
	PSPACE-hard Tiuryn [Tiu92]	
regular or infinite trees	in DEXPTIME Tiuryn,Wand [TW93]	DEXPTIME-complete [this thesis]
	DEXPTIME-hard [this thesis]	

Table 1.2: Complexity of subtype satisfiability with constants ordered in a poset. All stated results are proven in this thesis.

Tables 1.1 and 1.2 summarize complexity results regarding subtype satisfiability over lattices and posets. In particular, we show that recursive structural satisfiability is *DEXPTIME*-hard, finite non-structural satisfiability is *PSPACE*-complete, and recursive non-structural satisfiability is *DEXPTIME*-complete. This settles a longstanding problem left open by Tiuryn and Wand in 1993 [TW93].

1.3.2 Subtype entailment

Understanding the algorithmic properties of subtype entailment remains challenging. Even for weak type languages designing an efficient algorithm turned out to be surprisingly difficult; Table 1.3 summarizes the state of the art. The structural case was clarified by Rehof [Reh98]. For a restricted lattice of two ordered constants Rehof and Henglein showed that structural subtype entailment is coNP-complete [HR97] for finite trees (simple types) and PSPACE-complete [HR98] for regular or infinite trees.

Despite extensive efforts over many years, even the decidability of non-structural subtype entailment (NSSE) has been a prominent open problem in programming language theory. Only a PSPACE lower bound is known which holds in both cases, for finite and infinite trees [HR98]. The signature $\{\bot, \times, \top\}$ is enough to prove PSPACE hardness. Yet this result does not explain why finding a decision procedure for NSSE is so difficult. On the other hand, only a fragment of NSSE could be proved decidable [NP99] (and PSPACE-complete).

Our contribution to entailment yields a new characterization of NSSE that uses regular expressions and word equations [Mak77, Pla99]. More precisely we map NSSE to the question whether so called cap set expressions do contain all words over an alphabet

	structural *ordering over* $\{int, real, \times, \rightarrow\}$	*non-structural* *ordering over* $\{\bot, \top, \times\}$
finite trees	coNP-complete Henglein, Rehof [HR97]	universality of cap set expressions [this thesis]
regular or infinite trees	PSPACE-complete Henglein, Rehof [HR98]	universality of cap set expressions [this thesis]

Table 1.3: Complexity of subtype entailment.

(universality problem). Cap set expressions extend regular expressions R by new operators accepting the prefix closure of non-regular sets

$$\bigcup \{\ \pi^* \mid \pi \in R\ \}.$$

The universality problem of cap set expressions (Table 1.3) is reducible to the positive $\forall\exists^*$ fragment of the first-order theory of word equations with regular constraints. Unfortunately, even the positive $\forall\exists^3$ fragment of a single word equation is undecidable [Dur95] except if the alphabet is infinite [BS96] or a singleton [VR83]. Therefore, it remains open whether NSSE is decidable or not.

To narrow the problem, we translate the universality of certain cap expressions also back to NSSE. It becomes clear that the difficulty is raised by word equations hidden behind cap set expressions which spoil the usual pumping arguments from automata theory. They also clarify why NSSE differs so significantly from seemingly similar entailment problems [MNT98, NMT99].

Our characterization constitutes a promising starting point to further investigate decidability of NSSE. For instance, we can infer two new decidability results for subclasses of NSSE where $\times$ is refined to be monadic or the corresponding cap expressions of NSSE are built over restricted regular expressions.

1.4 Related work

1.4.1 Automata theory for subtyping

We motivate our and related automata approaches for subtype satisfiability and entailment.

Automata theory has been successfully used and partly developed to describe computational aspects of logical languages. Its key result is to correlate satisfiability of some

(weak) second-order monadic formula φ to emptiness of some appropriate automaton $\mathcal{A}_\varphi$ [Büc60, Nau66, Rab69, TW69].

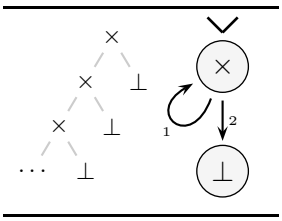
The regular part of our entailment characterization is based on a similar idea given by Kozen, Palsberg, and Schwartzbach [KPS95]. They apply finite automata theory to improve a classical DEXPTIME result of Amadio and Cardelli [AC93]: whether two mu terms are in the subtype relation. This problem is defined as non-structural satisfiability restricted to only one variable free literal $t_1 \leq t_2$ where t_1, t_2 are in a subtype language enriched by a mu operator. Its semantic is given by nested substitutions $[./.]$:

$$\mu x.\, t \;=\; t[x/\mu x.\, t]$$

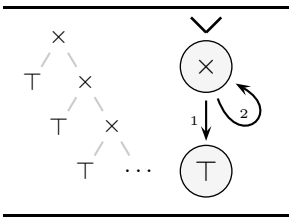
which describes an infinite unfolding of t. We consider for example the valid subtyping

$$\mu y.(y \times \bot) \leq \mu y.(\top \times y).$$

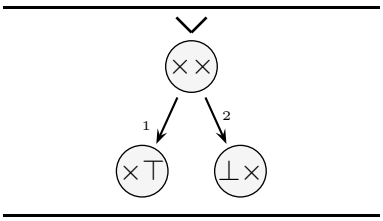
Kozen and Palsberg's approach first represents solutions of the left and right mu term by two finite automata (Figure 1.6 and 1.7) and second builds the product automaton of both (Figure 1.8). Subtyping applies if the product automaton contains no node where the left label is greater than the right label assuming $\bot \leq \times \leq \top$.



1.6. Regular tree and corresponding finite automaton satifying $\mu y.(y \times \bot)$.



1.7. The same for $\mu y.(\top \times y)$.



1.8. Subtyping is reduced to product automaton.

A similar automata theoretical approach is also known for the more general problem of subtype satisfiability [KPS94, PWO97].

We recently proposed an alternative tree automata based approach to non-structural subtype entailment [SAN+02]. It is completely unrelated to the present approach where we only deal with word automata. Tree automata are used in the alternative proposal of recognizing the set of all solutions of a subtype constraint. Every solution is seen as tuples of trees [CDG+02] that is recognized by tuple tree automata with equality constraints. But unfortunately, the emptiness problem of tuple tree automata with equality constraints is undecidable [SAN+, Tre00].

1.4.2 Constrained types

Corresponding to polymorphic type schemes in ML, polymorphic subtype systems could be expressed by constrained types, *i.e.*, types which are restricted by a set of subtype constraints [AW93, EST95a, EST95b]. Decidability of the question of whether or not one constrained type is a subtype of another one is a prominent open problem [Su02], a decidable approximation was presented in [TS96]. This and related open questions on constrained types can be reduced to the $\exists\forall^*$ fragment of the first-order theory over subtype constraints.

1.4.3 First-order theory over subtype constraints

The question of whether the first-order theory of subtype constraints is decidable or not, naturally extends our question on entailment's decidability. We have shown that the first-order theory of non-structural subtype constraints is undecidable via a reduction from Post's Correspondence Problem; this is shown for all three models: finite, regular, and infinite trees. It was shown recently that the first-order theory of structural subtyping over finite trees is decidable [KR03]. The decidability of the full first-order theory of structural subtyping over regular trees is still open.

	structural *ordering over* $\{int, real, \times, \rightarrow, ...\}$	*non-structural* *ordering over* $\{\bot, \top, \times\}$
finite trees	decidable Kuncak, Rinard [KR03]	undecidable Su,Aiken,Niehren,Priesnitz,Treinen [SAN$^+$02]
regular or infinte trees	open (at least non-elementary) Kuncak, Rinard [KR03]	undecidable Su,Aiken,Niehren,Priesnitz,Treinen [SAN$^+$02]

Table 1.4: Decidability of subtype first order theory.

1.4.4 Partial types

The presence of $\bot$ complicates type inference in contrast to the present of $\top$ which could even simplify type inference (see [Pie02] for a discussion). For this reason Thatte [Tha94] introduced partial types: a subtype language which supports $\top$ but not $\bot$. Partial type inference is considered for various calculi including Abadi and Cardelli's object calculus [AC96] and system F equipped with bounded polymorphism [CMMS94], a feature which arises when polymorphism and subtyping are combined.

Thatte gives a semi-decision procedure for type inference and satisfiability for finite trees in the absence of any constant besides $\top$. Decidability for satisfiability of partial

types is shown in [KPS94] for finite trees and in [OW92] for infinite trees. A process of closure and consistency checking proves both problems in P.

In the case of infinite trees and constants ordered in a lattice, Jim and Palsberg [JP99] found a new generic approach which subsumes complexity bounds for structural, non-structural subtyping (second line in Table 1.1) and partial types. Their idea is to refine the subtype relation by simulations, an idea from concurrency theory [Mil99]. They do so by:

$$\leq =_{\mathrm{df}} \bigcup\{\, R \mid R \text{ is a simulation} \,\}.$$

Using the principle of co-induction this leads to a very elegant polynomial algorithm for satisfiability of subtypes and partial types over several constant orderings.

So far the problem of entailment of partial types has not been addressed.

1.5 Publications

The material presented in this thesis has been published except of the new decidability result in the last chapter which remains to be unpublished. In the following we summarize all publications.

The first three publication of the author develop the results that characterize non-structural subtype entailment:

- with Joachim Niehren. Non-structural subtype entailment in automata theory. *Information and Computation*, 186(2):319–354, 2003.
- with Joachim Niehren. Non-structural subtype entailment in automata theory. In *International Symposium on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science 2215, page 360–384. Springer-Verlag, 2001.
- with Joachim Niehren. Entailment of non-structural subtype constraints. In *Asian Computer Science Conference*, Lecture Notes in Computer Science 1742, pages 251–265. Springer-Verlag, 1999.

The next two publications contain the results on subtype satisfiability over posets:

- with Joachim Niehren and Zhendong Su. Complexity of subtype satisfiability over posets. In *European Symposium on Programming*, Lecture Notes in Computer Science. Springer-Verlag, 2005. To appear.
- with Joachim Niehren. Satisfiability of structural subtype constraints. In *International Workshop on Unification*, pages 79–80, (best student award paper). Universidad Politécnica de Valencia, 2003.

Two further publications report results of the author on the first-order theory of non-structural subtype constraints that are not elaborated in this thesis.

- with Zhendong Su, Alexander Aiken, Joachim Niehren, and Ralf Treinen. First-order theory of subtyping constraints. *ACM Transactions on Programming Languages and Systems*. To appear 2005.
- with Zhendong Su, Alexander Aiken, Joachim Niehren, and Ralf Treinen. First-order theory of subtyping constraints. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 203–216, ACM Press, 2002.

2 Subtype Constraints

2.1 Basic concepts

We start with fundamental preliminaries for this work: finite, regular, and infinite trees over Σ. Finite trees model simple types while regular or infinite trees model recursive types. We use a standard definition of trees, whose idea is to identify every node of a tree with the word that addresses it relative to the root.

A *word* over an alphabet A is a finite sequence of letters in A. We denote words by π, μ, or ν and the set of words over A with A^*. The *empty word* is written as ε and the free-monoid *concatenation* of words π and μ by juxtaposition $\pi\mu$, with the property that $\varepsilon\pi = \pi\varepsilon = \pi$. A *prefix* of a word π is a word μ for which there exists a word ν such that $\pi = \mu\nu$. If μ is a prefix of π then we write $\mu \leq \pi$ and if μ is a proper prefix of π then we write $\mu < \pi$.

Types can be viewed as trees over some ranked alphabet Σ, the *signature* of the given type language. A signature consists of a finite set of function symbols (also called type constructors). Each function symbol f has an associated $\mathit{arity}(f) \geq 0$, indicating the number of arguments that f expects, and for all $1 \leq i \leq \mathit{arity}(f)$ a polarity $\mathit{pol}(f, i) \in \{1, -1\}$. We call a position i of symbol f *covariant* if $\mathit{pol}(f, i) = 1$ and *contravariant* otherwise. Symbols with arity zero are *type constants*.

We consider *trees* over Σ as partial functions $\tau : \mathbb{N}^* \rightsquigarrow \Sigma$ which map words over natural numbers to function symbols. The words in $D_\tau \subseteq \mathbb{N}^*$ are called the *nodes* or *paths* of the tree. We require that every tree has a root $\varepsilon \in D_\tau$ and that tree domains D_τ are always prefix closed and arity-consistent. The latter means for all trees τ, nodes $\pi \in D_\tau$, and naturals $i \in \mathbb{N}$ that $\pi i \in D_\tau$ if and only if $1 \leq i \leq \mathit{arity}(\tau(\pi))$. A tree τ is *finite* if its domain D_τ is finite and otherwise *infinite*. We write tree_Σ for the set of possibly infinite trees over Σ.

We call τ' the subtree of τ at path π if $\tau(\pi\pi') = \tau'(\pi')$ holds for all $\pi \in D_{\tau'}$. We write $\tau.\pi$ for the subtree of τ at node π under the presupposition $\pi \in D_\tau$. A tree is *regular* if it has at most finitely many distinct subtrees.

We will freely interpret function symbols in Σ as tree constructors. To make clear distinctions, we will write $=_\Sigma$ for equality of symbols in Σ and $=$ for equality of trees

over Σ. Given $g \in \Sigma$ and trees $\tau_1, \ldots, \tau_{arity(g)}$ we define $\tau = g(\tau_1, \ldots, \tau_{arity(g)})$ by $\tau(\varepsilon) =_\Sigma g$ and $\tau(i\pi) =_\Sigma \tau_i(\pi)$ for all $\pi \in D_{\tau_i}$ and $1 \leq i \leq arity(g)$. We thus consider ground terms over Σ as (finite) trees, for instance $f(\bot, \top)$ or $\bot$. Thereby, we have overloaded our notation since a constant $a \in \Sigma$ can also be seen as the tree τ with $\tau(\varepsilon) = a$. But this should never lead to confusion.

We define the *polarities* of nodes in trees as follows:

$$\begin{array}{lll} pol_\tau(\varepsilon) & =_{\text{df}} & 1 \\ pol_{f(\tau_1,\ldots,\tau_n)}(i\pi) & =_{\text{df}} & pol(f,i) * pol_{\tau_i}(\pi) \end{array}$$

2.2 Subtype orders

Subtype orders $\leq$ are partial orders on trees over some signature Σ. Two subtype orders arise naturally, *structural subtyping* and *non-structural* subtyping.

2.2.1 Structural subtyping

We investigate structural subtyping over standard signatures with posets. These are parametrized by posets $(B, \leq_B)$ of constants and have the form: $\Sigma = B \cup \{\times, \rightarrow\}$. The product type constructor $\times$ is a binary function symbol that is covariant in both positions ($pol(\times, 1) = pol(\times, 2) = 1$), while the function type constructor $\rightarrow$ is contravariant in its first and covariant in its second argument ($pol(\rightarrow, 1) = -1$ and $pol(\rightarrow, 2) = 1$).

Structural subtype orders $\leq$ are partial orders on trees over structural signatures Σ. They are obtained by lifting the ordering on constants $(B, \leq_B)$ in Σ to trees. More formally, $\leq$ is the smallest binary relation $\leq$ on $tree_\Sigma$ such that for all $b, b' \in B$ and types τ_1, τ_2, τ_1', τ_2' in $tree_\Sigma$:

- $b \leq b'$ iff $b \leq_B b'$;
- $\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'$ iff $\tau_1 \leq \tau_1'$ and $\tau_2 \leq \tau_2'$;
- $\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'$ iff $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$.

Notice that $\times$ is monotonic in both of its arguments while $\rightarrow$ is anti-monotonic in its first argument and monotonic in its second. For more general signatures, monotonic arguments are specified by covariant positions of function symbols, and anti-monotonic arguments by contravariant positions.

For structural subtyping, two types are related only if they have exactly the same shape, *i.e.*, tree domain. Notice that structural subtype orders are indeed partial

orders. We do not restrict ourselves to lattices $(B, \leq_B)$ in contrast to most previous work.

2.2.2 Non-structural subtyping

In the *non-structural subtype order*, two distinguished constants are added to structural type languages, a *smallest type* $\bot$ and a *largest type* $\top$. The ordering is parametrized by a poset $(B, \leq_B)$ and has the signature: $\Sigma = B \cup \{\times, \rightarrow\} \cup \{\bot, \top\}$. For the non-structural subtype order, besides the three structural rules earlier, there is an additional rule: $\bot \leq \tau \leq \top$ for any $\tau \in \mathit{tree}_\Sigma$. Again $\leq$ is a reflexive, transitive, and anti-symmetric relation and thus, a partial order.

2.2.3 Uniform subtyping

We introduce *uniform subtyping* as an intermediate ordering for two reasons: (i) to capture both structural and non-structural subtyping effects and (ii) to use it as a bridge from uniform subtype constraints to modal logic.

We call a signature Σ *uniform* if all symbols in Σ have the same non-zero arity and the same polarities. All trees over Σ are complete infinite n-ary trees, where n is the arity common to all function symbols in Σ. Hence, all trees have the same shape. Furthermore, the polarities of nodes $\pi \in \{1, \ldots, n\}^*$ in trees τ over uniform signatures do not depend on τ. We therefore write $\mathit{pol}(\pi)$ instead of $\mathit{pol}_\tau(\pi)$.

The signatures $\{\times\}$ and $\{\rightarrow\}$, for instance, are both uniform, while $\{\times, \rightarrow\}$ or $\{\bot, \top, \times\}$ are not. The idea to model the non-structural signature $\{\bot, \top, \times\}$ uniformly is to raise the arities of $\bot$ and $\top$ to 2 and to order them by $\bot \leq_\Sigma \times \leq_\Sigma \top$.

A *uniform subtype order* $\leq$ is defined over a partially-ordered uniform signature $(\Sigma, \leq_\Sigma)$. It satisfies for all trees $\tau_1, \tau_2 \in \mathit{tree}_\Sigma$:

$$\tau_1 \leq \tau_2 \quad \text{iff} \quad \forall \pi \in \{1, \ldots, n\}^* : \ \tau_1(\pi) \leq_\Sigma^{\mathit{pol}(\pi)} \tau_2(\pi)$$

where n is the arity of the function symbols in Σ. For simplicity, we will often write $\leq_\Sigma^\pi$ instead of $\leq_\Sigma^{\mathit{pol}(\pi)}$.

2.3 Subtype constraints

In a subtype system, type variables are used to denote unknown types. We assume an infinite set of tree valued type variables that we denote by x, y, z, u, v, w. A *s*ubtype constraint φ is a conjunction of literals with the following abstract syntax:

$$\varphi \ ::= \ x{=}f(x_1, \ldots, x_n) \mid x{\leq}y \mid \varphi \wedge \varphi$$

where n is the arity of $f \in \Sigma$. We interpret constraints φ in the structure of trees over Σ with non-structural subtyping. We distinguish three cases, the structure of finite trees, of regular trees, or else of possibly infinite trees. We interpret function symbols in both cases as tree constructors and the predicate symbol $\leq$ by the non-structural subtype relation. Again, this overloads notation: we use the same symbol $\leq$ for the subtype relation on trees and the predicate symbol denoting the subtype relation in constraints. Again, this should not raise confusion.

We also use an alternative subtype constraint language in the first Chapter 3 of Part I and in all chapters of Part II where we forbid formulas $x{\leq}y$:

$$\varphi \ \ ::= \ \ x{\leq}f(y_1, \ldots, y_n) \mid f(y_1, \ldots, y_n){\leq}x \mid x{=}c \mid \varphi \wedge \varphi'$$

where $n \neq 0$ is the arity of a non-constant $f \in \Sigma$ and $c \in \Sigma$ is a constant. This choice will help us to simplify our presentation essentially. It is, however, irrelevant from the point of view of expressiveness. We can still express $x{\leq}y$ by using existential quantifiers:

$$x{\leq}y \leftrightarrow \exists z \exists u \ (f(x, u, \ldots, u){\leq}z \wedge z{\leq}f(y, u, \ldots, u))$$

As in this equivalence, we will sometimes use first-order formulas Φ built from constraints and the usual first-order connectives. We will write $V(\Phi)$ for the set of free variables occurring in Φ.

2.4 Subtype satisfiability

A solution of Φ is a variable assignment α into the set of finite (resp. regular or possibly infinite) trees which satisfies the required subtype relations; we write $\alpha \models \Phi$ if α solves Φ and say that Φ is *satisfiable*.

EXAMPLE. The constraint $x{\leq}f(x)$ is satisfiable, even when interpreted over finite trees. We can solve it by mapping x to $\bot$. In contrast, the equality constraint $x{\leq}f(x) \ \wedge \ f(x){\leq}x$ is unsatisfiable over finite trees. It can however be solved by mapping x to the infinite tree $f(f(f(\ldots)))$.

We distinguish three subtype satisfiability problems, each of which has various variants depending on interpretation over finite, regular, or possibly infinite trees and constants ordered in a lattice or a poset.

Structural subtype satisfiability is the problem to decide whether a structural subtype constraint is satisfiable. The arguments of this problem are a lattice or posets $(B, \leq_B)$ and a constraint φ over the signature $B \cup \{\times, \to\}$.

Non-structural subtype satisfiability is the problem to decide whether a non-structural subtype constraint is satisfiable. The arguments are a lattice or poset $(B, \leq_B)$ and a constraint φ over signature $B \cup \{\times, \to\} \cup \{\bot, \top\}$.

Uniform subtype satisfiability is the the problem to decide whether a uniform subtype constraint is satisfiable. The arguments are a partially-ordered uniform signature $(\Sigma, \leq_\Sigma)$ and a subtype constraint φ over this signature.

2.5 Subtype entailment

A formula Φ_1 *entails* Φ_2 (we write $\Phi_1 \models \Phi_2$) if all solutions $\alpha \models \Phi_1$ satisfy $\alpha \models \Phi_2$. We will consider entailment judgments that are triples of the form (φ, x, y) that we write as $\varphi \models^? x \leq y$.

Non-structural subtype entailment (NSSE) is the problem to decide whether entailment $\varphi \models x \leq y$ holds. The argument is an entailment judgment $\varphi \models^? x \leq y$ over a restricted signature $\{f, \bot, \top\}$ with a single non-constant function symbol f that is covariant.

The choice of such signatures imposes two restrictions: first, we do not allow for contravariant type constructors. These could be covered in our framework even though this is not fully obvious. Second, we do not treat larger signatures with more than one non-constant function symbol or other constants beside $\bot$ and $\to$. This is a true restriction that cannot be circumvented easily.

Note that entailment judgments of the simple form $\varphi \models^? x \leq y$ can express general entailment judgments, where both sides are conjunctions of inequations $t_1 \leq t_2$ between nested terms or variables (i.e. $t ::= x \mid f(t_1, \ldots, t_n) \mid \bot \mid \top$). The main trick is to replace a judgment $\varphi \models^? t_1 \leq t_2$ with terms t_1 and t_2 by $\varphi \wedge x = t_1 \wedge y = t_2 \models^? x \leq y$ where x and y are fresh variables. Note also that the omission of formulas $u \leq v$ on the left hand side does not restrict the problem. (Existential quantifier on the left hand side of an entailment judgment can be removed.)

EXAMPLE. The prototypical example where NSSE holds somehow surprisingly is:

$$x{\leq}f(y) \wedge f(x){\leq}y \models^{?} x{\leq}y \qquad \text{(yes)}$$

To see this, note that all finite trees in the unary case are of the form $f \ldots f(\bot)$ or $f \ldots f(\top)$. Thus, $x{\leq}y \vee y{<}x$ is valid in this case. Next let us contradict the assumption that there is a solution $\alpha \models y{<}x \wedge x{\leq}f(y) \wedge f(x){\leq}y$. Transitivity yields $\alpha(y){\leq}f(\alpha(y))$ and then also $f(\alpha(x)){\leq}f(\alpha(y))$. Hence $\alpha(x){\leq}\alpha(y)$ which contradicts $\alpha(y){<}\alpha(x)$.

Part I

Subtype Satisfiability

Satisfiability

3 Subtyping over a Lattice

4 Subtyping over a Poset

For non-structural subtype satisfiability over a lattice we repeat a well-known closure and consistency test. In a second step we extend it by a first-order unification test to obtain a procedure for structural subtype satisfiability.

We prove all complexities of subtype satisfiability over posets by connecting subtype constraints and modal logic. For the regular or infinite case we choose a dialect of propositional dynamic logic which we proved to be DEXPTIME-complete by automata theory; in case of finite trees we choose a sublanguage of K-normal modal logic which we prove to be PSPACE-complete by an idea of Spaan [Spa93, Hem00].

3 Subtyping over a Lattice

We presents satisfiability tests for structural and non-structural subtype constraints where we consider the two cases of possibly infinite and resp. regular trees first and then turn to the third and more difficult case of finite trees. Satisfiability of infinite trees was studied by Palsberg and O'Keefe [PO95], and additionally by Pottier [Pot96]. The case of finite trees was solved by Palsberg, Wand, and O'Keefe [PWO97]. An earlier approach [KPS94] treats the simpler problem of partial types which includes a greatest but no least type $\bot$. As they do, we will construct solutions with smallest shape for satisfiable constraints. In the infinite and regular cases, we will construct least and greatest solutions. These will prove extremely useful for proving the completeness of our non-structural entailment test in the next part of this thesis.

To simplify our presentation we omit the contravariant function symbol $\rightarrow$ in the signature during this chapter; all given proofs would also work in the present of $\rightarrow$ but require a more elaborated form of syntactic support which based on polarities.

3.1 Non-structural subtyping over infinite trees

Let φ be a subtype constraint over a signature $\{\bot, \top, f\} \cup B$ where B is a lattice of constants, *i.e.* a partial order in which every set has a least upper and greatest lower bound.

reflexive	$\varphi \vdash x \leq x$	for variables $x \in V(\varphi)$
trans.	$\varphi \vdash x \leq z$	if $\varphi \vdash x \leq y$ and $\varphi \vdash y \leq z$
decomp.	$\varphi \vdash x_i \leq y_i$	if $1 \leq i \leq n$, $\varphi \vdash x \leq y$, $f(x_1, \ldots, x_n) \leq x \wedge y \leq f(y_1, \ldots, y_n)$ in φ

Table 3.1: Syntactic support of inequalities.

$\varphi \vdash x{\leq}i(z)$	if $\varphi \vdash x{\leq}x'$, $x'{\leq}f(y_1,\ldots,y_n)$ in φ, $\varphi \vdash y_i{\leq}z$, and $1 \leq i \leq \mathit{arity}(f)$
$\varphi \vdash i(z){\leq}x$	if $\varphi \vdash z{\leq}y_i$, $f(y_1,\ldots,y_n){\leq}x'$ in φ, $\varphi \vdash x'{\leq}x$, and $1 \leq i \leq \mathit{arity}(f)$

Table 3.2: Syntactic support of lower and upper bounds for i-th children.

3.1.1 Closure algorithm

As a prerequisite for our satisfiability and entailment test, we use a closure algorithm. This algorithm computes a set of inequalities of the form $x{\leq}y$ that are syntactically supported by a constraint φ.

In contrast to other closure algorithms, we cannot simply add supported inequalities to the initial constraint φ. The reason is that for our later entailment characterization we are restricted to use the second defined subtype language in the last Chapter 2:

$$\varphi \quad ::= \quad x{\leq}f(y_1,\ldots,y_n) \mid f(y_1,\ldots,y_n){\leq}x \mid x{=}c \mid \varphi \wedge \varphi'$$

where $n \neq 0$ is the arity of a non-constant $f \in \Sigma$ and $c \in \Sigma$ is a constant.

Instead, Table 3.1 defines judgments $\varphi \vdash x{\leq}y$ which state that φ supports $x{\leq}y$ syntactically. The definition consists of three standard rules. The first two rules express the reflexivity and transitivity of the subtype ordering. Finally and most importantly, the definition of syntactic support accounts for decomposition which can be applied recursively.

To keep our proofs simple, Table 3.2 define two further forms of syntactic support: $\varphi \vdash x{\leq}i(y)$ means that y is an upper bound of the i-th child of x and $\varphi \vdash i(y){\leq}x$ states the symmetric lower bound for x at i.

Lemma 3.1
For all φ, x, y, and $1 \leq i \leq n = \mathit{arity}(f)$ if $\varphi \vdash x{\leq}i(z)$ then:

$$\varphi \models \exists y_1 \ldots \exists y_{i-1} \exists y_{i+1} \ldots \exists y_n.\ x{\leq}f(y_1,\ldots,y_{i-1},z,y_{i+1},\ldots,y_n).$$

The symmetric property for lower bound $\varphi \vdash i(z){\leq}x$ is valid too.

PROOF. Obvious from Lemma 3.2 and Table 3.2.

C1.	$\varphi \vdash x{\leq}y$ and $x{=}c_1 \wedge y{=}c_2$ in φ	for $c_1, c_2 \in B \cup \{\bot, \top\} : c_1 >_\Sigma c_2$
C2.	$\varphi \vdash x{\leq}y$ and $x{=}c \wedge y{\leq}f(y_1, \ldots, y_n)$ in φ	for $c \in B \cup \{\top\}$
C3.	$\varphi \vdash x{\leq}y$ and $f(x_1, \ldots, x_n){\leq}x \wedge y{=}c$ in φ	for $c \in B \cup \{\bot\}$

Table 3.3: Label clashes (the definition of judgments $\varphi \vdash x{\leq}y$ is in Table 3.1).

3.1.2 Closure clash

A label clash in a φ imposes an unsatisfiable condition on the root label of some tree described by φ. Table 3.3 collects three kinds of label clashes. Clash *C1* requires two constants c_1, c_2 with $c_1{\leq}c_2$ and $c_1 >_\Sigma c_2$, clash *C2* needs $c{\leq}f$ for some $c{\neq}\bot$, and *C3* imposes $f{\leq}c$ for some $c{\neq}\top$.

Lemma 3.2
For all φ, x, and y: if $\varphi \vdash x{\leq}y$ then $\varphi \models x{\leq}y$.

PROOF. By induction on the definition of syntactic support of inequalities.

Lemma 3.3
If φ contains a label clash then it is unsatisfiable over possibly infinite trees (and thus also over regular or finite trees).

PROOF. We only consider the case where φ contains a label clash of form *C1*. The two remaining cases *C2* and *C3* are analogous. If φ contains a label clash of form *C1* then:

$$x{=}c_1 \text{ in } \varphi,\ y{=}c_2 \in \varphi, \text{ and } \varphi \vdash x{\leq}y$$

for some variables x, y in $V(\varphi)$ and constants $c_1 >_\Sigma c_2$. Lemma 3.2 yields $\varphi \models x{\leq}y$. Thus, $\varphi \models c_1{\leq}c_2$ which requires that φ is unsatisfiable.

Proposition 3.1
A constraint is satisfiable over the structure of possibly infinite trees if and only if it does not contain any label clash (see Table 3.3).

One direction of Proposition 3.1 coincides with Lemma 3.3. The converse is proved below.

In order to construct least or greatest solutions, we need the notions of lower and upper bounds in subtree positions. Let u, v be variables and $\pi \in \{1, \ldots, \mathit{arity}(f)\}^*$

$\varphi \vdash x{\leq}\varepsilon(y)$	if $\varphi \vdash x{\leq}y$
$\varphi \vdash \varepsilon(x){\leq}y$	if $\varphi \vdash x{\leq}y$
$\varphi \vdash x{\leq}\pi i(y)$	if $\varphi \vdash x{\leq}\pi(z),\ z{\leq}f(z_1, \ldots z_i \ldots, z_n)$ in $\varphi,\ \varphi \vdash z_i{\leq}y$
$\varphi \vdash \pi i(x){\leq}y$	if $\varphi \vdash \pi(z){\leq}y,\ f(z_1, \ldots z_i \ldots, z_n){\leq}z$ in $\varphi,\ \varphi \vdash x{\leq}z_i$

Table 3.4: Syntactic support of lower and upper bounds.

paths. Let terms $u.\pi$ to denote the subtree of the value of u at node π under the presupposition of existence. We define two first-order formulas by:

$$\begin{array}{lll} u{\leq}\pi(v) & =_{def} & \exists w : u{\leq}w \wedge w.\pi = v \\ \pi(u){\leq}v & =_{def} & \exists w : w{\leq}v \wedge w.\pi = u \end{array}$$

The formula $u{\leq}\pi(v)$ means that v is an *upper bound* of u at node π. It is satisfied by variables assignments α where either $\alpha(u)(\pi') = \bot$ for some prefix π' of π or $\alpha(u).\pi \leq \alpha(v)$. Symmetrically, $\pi(u){\leq}v$ states that u is a *lower bound* of v at π. It is satisfied by variables assignments α where either $\alpha(v)(\pi') = \top$ for some prefix π' of π or $\alpha(u) \leq \alpha(v).\pi$.

Table 3.4 defines syntactic support of lower and upper bounds at subtree positions. This generalizes both notions of syntactic support for inequations and children positions in Tables 3.1 and 3.2. Note in particular that both judgments $\varphi \vdash x{\leq}\varepsilon(y)$ and $\varphi \vdash \varepsilon(x){\leq}y$ are equivalent to $\varphi \vdash x{\leq}y$.

Lemma 3.4
If $\varphi \vdash \pi(x){\leq}y$ then $\varphi \models \pi(x){\leq}y$ and if $\varphi \vdash y{\leq}\pi(x)$ then $\varphi \models y{\leq}\pi(x)$.

PROOF. By induction on the derivation of syntactic support. The base case relies on Lemma 3.1.

Lemma 3.5 (Decomposition)

1. *If $\varphi \vdash \pi(w){\leq}u$ and $\varphi \vdash u{\leq}\pi\pi'(v)$ then $\varphi \vdash w{\leq}\pi'(v)$.*
2. *If $\varphi \vdash \pi\pi'(v){\leq}u$ and $\varphi \vdash u{\leq}\pi(w)$ then $\varphi \vdash \pi'(v){\leq}w$.*

PROOF. By induction on the length of the path π.

PROOF(Proof of Proposition 3.1). We have to construct solutions for constraints φ that do not have label clashes. The basic idea is to construct the least solution by

mapping variables x of $V(\varphi)$ to the least upper bound of all lower bounds of x in φ. Symmetrically, we could also choose the greatest solution.

The construction will use additional judgments $\varphi \vdash \pi(g) \leq x$ where $g \in \Sigma$. Such judgments say that φ syntactically supports g to be a lower bound for the label of x at path π. It is defined as follows:

$$\begin{array}{ll} \varphi \vdash \pi(c) \leq x & \text{if } \varphi \vdash y{=}c \text{ in } \varphi \text{ and } \varphi \vdash \pi(y) \leq x \\ \varphi \vdash \pi(f) \leq x & \text{if } \varphi \vdash f(y_1, \ldots, y_n) \leq y \text{ in } \varphi \text{ and } \varphi \vdash \pi(y) \leq x \end{array}$$

Symmetric judgments of the forms $\varphi \vdash x \leq \pi(c)$ and $\varphi \vdash x \leq \pi(f)$ are defined in analogy.

We now specify variable assignments least_φ which map variables $z \in V(\varphi)$ to the least upper bound of all lower bounds of z in φ. We define $\text{least}_\varphi(z)(\pi)$ by induction on the length of π. Given a word π for which $\text{least}_\varphi(z)(\pi') \notin \{\bot, \top\}$ for all proper prefixes $\pi' < \pi$, we set:

$$\text{least}_\varphi(z)(\pi) = \sup\{\, g \mid \varphi \vdash \pi(g) \leq z \,\}$$

Otherwise, we leave $\text{least}_\varphi(z)(\pi)$ undefined. The definition is sound since all subsets of Σ have a least upper bound. Note also that $\bot$ is the least upper bound of the empty subset of Σ.

Clearly, $\text{least}_\varphi(z)$ is a tree over Σ: its domain is prefix closed and arity consistent by definition. Note that $\text{least}_\varphi(z)$ may be infinite, for instance, if φ is the constraint $f(z) \leq z$.

It remains to show that least_φ is indeed a solution of φ, i.e., that it satisfies all literals of φ. To prove this we distinguish all possible kinds of literals.

1. Case $z{=}\bot$ in φ. In this case: $\text{least}_\varphi(z)(\varepsilon) = \bot$. Otherwise, $\varphi \vdash \varepsilon(\top) \leq z$ or $\varphi \vdash \varepsilon(f) \leq z$. But φ then contains a label clash by *C1* or *C3* which contradicts our assumption.
2. Case $z{=}\top$ in φ. Obviously, $\text{least}_\varphi(z)(\varepsilon) = \top$.
3. Case $z{=}c$ in φ. Also obviously, $\text{least}_\varphi(z)(\varepsilon) = c$.
4. Case $f(z_1, \ldots, z_n) \leq z$ in φ. Let π be in the common domain of $\text{least}_\varphi(z)$ and $\text{least}_\varphi(f(z_1, \ldots, z_n))$.
   a) Case $\pi = \varepsilon$. Thus, $\varphi \vdash \varepsilon(f) \leq z$ so that the least solution satisfies $\text{least}_\varphi(z)(\varepsilon) \geq_\Sigma f = \text{least}_\varphi(f(z_1, \ldots, z_n))(\varepsilon)$.

b) Case $\pi = i\pi'$ for some $1 \leq i \leq n$. Note that $\text{least}_\varphi(f(z_1, \ldots, z_n))$ is equal to $f(\text{least}_\varphi(z_1), \ldots, \text{least}_\varphi(z_n))$ so that π' must belong to the domain of $\text{least}_\varphi(z_i)$ since π belongs to the domain of $\text{least}_\varphi(f(z_1,\ldots,z_n))$. This implies that $\text{least}_\varphi(z)(\pi')$ is defined and equal to $\sup\{g \mid \varphi \vdash \pi'(g) \leq z_i\}$. But if $\varphi \vdash \pi'(g) \leq z_i$ then $\varphi \vdash i\pi'(g) \leq z$, and hence:

$$\begin{array}{rcl} \text{least}_\varphi(z)(i\pi') & = & \sup\{\, g \mid \varphi \vdash \pi(g) \leq z \,\} \\ & \geq_\Sigma & \sup\{\, g \mid \varphi \vdash \pi'(g) \leq z_i \,\} \\ & = & \text{least}_\varphi(z_i)(\pi') \;= \text{least}_\varphi(f(z_1, \ldots, z_n))(i\pi') \end{array}$$

5. Case $z \leq f(z_1, \ldots, z_n)$ in φ. Let π be in the common domain of $\text{least}_\varphi(z)$ and $\text{least}_\varphi(f(z_1, \ldots, z_n))$.

   a) Case $\pi = \varepsilon$. Since φ does not contain any label clash of kind *C2*, it cannot hold that $\varphi \vdash \varepsilon(\top) \leq z$ and hence $\text{least}_\varphi(z)(\varepsilon) \leq_\Sigma f$.

   b) Case $\pi = i\pi'$ for some $1 \leq i \leq n$. Whenever $\varphi \vdash i\pi'(g) \leq z$ then $\varphi \vdash \pi'(g) \leq z_i$ by **decomposition** with Lemma 3.5. Hence:

$$\begin{array}{rcl} \text{least}_\varphi(z)(\pi) & = & \sup\{\, g \mid \varphi \vdash \pi'(g) \leq z \,\} \\ & \leq_\Sigma & \sup\{\, g \mid \varphi \vdash \pi'(g) \leq z_i \,\} \\ & = & \text{least}_\varphi(z_i)(\pi') \;=\; \text{least}_\varphi(f(z_1, \ldots, z_n))(i\pi') \end{array}$$

By inspection of the proof of Proposition 3.1 we obtain the following additional result on the existence and form of least and greatest solutions. This result is of its own relevance but also important with respect to entailment.

Proposition 3.2
Every constraints that is satisfiable over possibly infinite trees permits a least solution least_φ *and a greatest solution* great_φ *(over possibly infinite trees). These solutions satisfy for all variables* $z \in V(\varphi)$ *and nodes* $\pi \in D_{\text{least}_\varphi(z)}$ *resp.* $\pi \in D_{\text{great}_\varphi(z)}$*:*

$$\begin{array}{rcl} \text{least}_\varphi(z)(\pi) & = & \sup\{\, g \mid \varphi \vdash \pi(g) \leq z \,\} \\ \text{great}_\varphi(z)(\pi) & = & \inf\{\, g \mid \varphi \vdash z \leq \pi(g) \,\} \end{array}$$

PROOF. The proof of Proposition 3.1 shows that least_φ is indeed solutions of φ; and this solution is clearly smaller than all other solutions of φ. By symmetry, the result for great_φ follows.

3.1.3 Regular trees

Least solutions always map variables to regular trees. This has the following consequence:

Proposition 3.3
A subtype constraint is satisfiable over finite trees if and only if it is satisfiable over regular trees.

PROOF. Let φ be a constraint that is satisfiable over finite trees, i.e. which does not have any label clash. We show that the least solution least_φ of φ maps to regular trees only. Proposition 3.2 shows that the least solution least_φ satisfies:

$$\text{least}_\varphi(z)(\pi) = \sup\{\, g \mid \varphi \vdash \pi(g) \leq z \,\}$$

for all $z \in V(\varphi)$ where $\text{least}_\varphi(z)(\pi') \notin \{\bot, \top\}$ for all proper prefixes $\pi' < \pi$, and that $\text{least}_\varphi(z)(\pi)$ is undefined otherwise. We next fix a variable $z \in V(\varphi)$ and show that $\text{least}_\varphi(z)$ has only finitely many distinct subtrees. We define for all paths $\pi \in A_\Sigma$:

$$V(\varphi)(z, \pi) = \{\, y \mid \varphi \vdash \pi(y) \leq z \,\}$$

The following claim follows straightforwardly from the definition of least solutions. For all $\pi, \pi' \in D_{\text{least}_\varphi(z)}$:

$$\text{least}_\varphi(z).\pi = \text{least}_\varphi(z).\pi' \text{ if and only if } V(\varphi)(z, \pi) = V(\varphi)(z, \pi')$$

This claim implies that the number of distinct subtrees of $\text{least}_\varphi(z)$ is uniformly bounded for all $z \in V(\varphi)$ by the number of subsets of $V(\varphi)$, and this number is finite.

3.2 Non-structural subtyping over finite trees

Satisfiability becomes more tedious in the case of finite trees where we have to care about unsatisfiable cycles. Most typically, $x \leq f(x) \wedge f(x) \leq x$ is unsatisfiable while $f(x) \leq x$ is satisfiable over finite trees. These two examples illustrate that only two-sided cycles in upper and lower bounds are unsatisfiable of finite trees, while one-side cycles can be satisfied. The general form of cycle clashes is given by rule *C4* in Table 3.5.

Lemma 3.6
A constraint with cycle clash is unsatisfiable over finite trees.

C4.	$\exists \pi \neq \varepsilon : \ \varphi \vdash \pi(x) \leq x$ and $\varphi \vdash x \leq y$ and $\varphi \vdash y \leq \pi(y)$

Table 3.5: Cycle clashes.

To simplify the presentation we will consider no constants besides $\bot$ and $\top$ in the signature Σ. The most problematic fact about satisfiability over finite trees is that satisfiable constraints need not have least or greatest solutions, in contrast to possibly infinite or regular trees (Proposition 3.2). But fortunately, there always exist solutions with least shape (i.e. least tree domain) for all constraints without label clashes [PWO97]. And least-shape solutions are always finite for constraints without cycle clashes.

The constraint $f(x) \leq x$, for instance, does not have a least solution when interpreted over finite trees. Its finite solutions map x to some tree of the form $f(f(f(\ldots(\top)\ldots)))$, none of which is smaller than all others. The solution mapping of x to $\top$ has the least shape but is greater than all others. Over possible infinite trees, there exists a least solution which maps x to the infinite tree $f(f(f(\ldots)))$.

Proposition 3.4
A constraint φ is satisfiable over the structure of finite trees if and only if it does not contain a label clash nor a cycle clash.

We have already shown the correctness of our clash rules (Lemmas 3.3 and 3.6); it remains to prove that constraints without label and cycle clash are satisfiable. This will be the content of Lemmas 3.7 and 3.8.

Given a satisfiable constraint φ we now define an assignment s_φ which maps variables to trees with the least possible shape for satisfying φ, i.e., with the least possible tree domain. Let $z \in V(\varphi)$ and $\pi \in \{1, \ldots, \mathit{arity}(f)\}^*$. We define $\mathsf{s}_\varphi(z)(\pi)$ by induction on the length of π so that we can assume that $\mathsf{s}_\varphi(z)(\pi') = f$ for all proper prefixes of π' of π:

$$\mathsf{s}_\varphi(z)(\pi) = \begin{cases} f & \text{if } \varphi \vdash \pi(f) \leq z \text{ and } \varphi \vdash z \leq \pi(f) \\ \bot & \text{if } \varphi \nvdash \pi(f) \leq z \text{ and } \varphi \vdash z \leq \pi(g) \text{ where } g \in \{\bot, f\} \\ \top & \text{else} \end{cases}$$

Lemma 3.7
If φ is free of label clashes then $\mathsf{s}_\varphi \models \varphi$ over possibly infinite trees.

PROOF. We show that s_φ satisfies all literals of φ.

1. Case $z{=}\bot$ in φ. The second clause of the definition of $\mathsf{s}_\varphi(z)(\varepsilon)$ applies since $\varphi \vdash \varepsilon(f){\leq}z$ would prove a label clash of kind *C3* otherwise. Hence $\mathsf{s}_\varphi(z) = \bot$.

2. Case $z{=}\top$ in φ. The first two clauses of the definition of $\mathsf{s}_\varphi(z)(\varepsilon)$ cannot apply in the absence of label clashes of forms *C1* and *C2*. Hence, $\mathsf{s}_\varphi(z) = \top$.

3. Case $f(z_1,\ldots,z_n){\leq}z$ in φ. Let π be in the common domain of $\mathsf{s}_\varphi(z)$ and $\mathsf{s}_\varphi(f(z_1,\ldots,z_n))$. Note that $\mathsf{s}_\varphi(f(z_1,\ldots,z_n)) = f(\mathsf{s}_\varphi(z_1),\ldots,\mathsf{s}_\varphi(z_n))$. It remains to prove $\mathsf{s}_\varphi(f(z_1,\ldots,z_n))(\pi) \leq_\Sigma \mathsf{s}_\varphi(z)(\pi)$.

   a) Subcase $\pi = \varepsilon$. Since $f(z_1,\ldots,z_n){\leq}z$ in φ, $\mathsf{s}_\varphi(z)(\varepsilon)$ cannot be defined by the second clause, and thus $\mathsf{s}_\varphi(z)(\varepsilon) \geq_\Sigma f$.

   b) Subcase $\pi = i\pi'$ for some $1 \leq i \leq n$. We must prove $\mathsf{s}_\varphi(z_i)(\pi') \leq_\Sigma \mathsf{s}_\varphi(z)(\pi)$ under the assumption that both values are defined. Clearly, this holds in the case $\mathsf{s}_\varphi(z)(\pi) = \top$. If $\mathsf{s}_\varphi(z)(\pi) =_\Sigma f$ then $\varphi \vdash z{\leq}\pi(f)$ so that the decomposition Lemma 3.5 yields $\varphi \vdash z_i{\leq}\pi'(f)$. Hence, $\mathsf{s}_\varphi(z_i)(\pi') \leq_\Sigma f = \mathsf{s}_\varphi(z)(\pi)$. Otherwise, $\mathsf{s}_\varphi(z)(\pi) = \bot$ so that $\varphi \not\vdash \pi(f){\leq}z$ and $\varphi \vdash z{\leq}\pi(g)$ for some $g \in \{f, \bot\}$. Hence, $\varphi \not\vdash \pi'(f){\leq}z_i$ and $\varphi \vdash z_i{\leq}\pi'(g)$. This implies $\mathsf{s}_\varphi(z_i)(\pi') = \bot$.

4. Case $z{\leq}f(z_1,\ldots,z_n)$ in φ. For given a word π in the domains of $\mathsf{s}_\varphi(z)$ and $\mathsf{s}_\varphi(f(z_1,\ldots,z_n))$ we prove $\mathsf{s}_\varphi(z)(\pi) \leq_\Sigma \mathsf{s}_\varphi(f(z_1,\ldots,z_n))(\pi)$.

   a) Subcase $\pi = \varepsilon$. Since $z{\leq}f(z_1,\ldots,z_n)$ in φ one of the first two clauses defines $\mathsf{s}_\varphi(z)(\varepsilon)$ so that $\mathsf{s}_\varphi(z)(\varepsilon) = f$ as required.

   b) Subcase $\pi = i\pi'$ for some $1 \leq i \leq n$. We show $\mathsf{s}_\varphi(z)(\pi) \leq_\Sigma \mathsf{s}_\varphi(z_i)(\pi')$. This clearly holds if $\mathsf{s}_\varphi(z_i)(\pi') = \top$. If $\mathsf{s}_\varphi(z_i)(\pi') = \bot$ then $\varphi \not\vdash \pi'(f){\leq}z_i$ and $\varphi \vdash z_i{\leq}\pi'(g)$ for some $g \in \{\bot, f\}$. Hence, $\varphi \not\vdash \pi(f){\leq}z$ according to the decomposition Lemma 3.5. Furthermore, $\varphi \vdash z{\leq}\pi(g)$ so that $\mathsf{s}_\varphi(z)(\pi) = \bot$. The final case is $\mathsf{s}_\varphi(z_i)(\pi') = f$. Now, $\varphi \vdash z_i{\leq}\pi'(f)$ and also $\varphi \vdash \pi'(f){\leq}z_i$. Thus, $\varphi \vdash z{\leq}\pi(f)$ so that $\mathsf{s}_\varphi(z)(\pi){\leq_\Sigma}f$.

Lemma 3.8
If φ doesn't contain cycle clashes then $\mathsf{s}_\varphi(z)$ is finite for all $z \in V(\varphi)$.

PROOF. Suppose that $\mathsf{s}_\varphi(z)$ is infinite for some $z \in V\varphi$. Then there exists an infinite word ω over the alphabet $\{1,\ldots,\mathit{arity}(f)\}$ such that $\mathsf{s}_\varphi(z)(\pi) = f$ for all finite prefixes π of ω. This yields $\varphi \vdash \pi(f){\leq}z$ and $\varphi \vdash z{\leq}\pi(f)$ for all such prefixes. Hence, there exists variables $x_\pi, y_\pi \in V\varphi$ for all prefixes π of ω such that $z = x_\varepsilon = y_\varepsilon$ and for all prefixes $\pi\pi'$ of ω:

$$\varphi \vdash \pi'(x_{\pi\pi'}){\leq}x_\pi \qquad \text{and} \qquad \varphi \vdash x_\pi{\leq}y_\pi \qquad \text{and} \qquad \varphi \vdash y_\pi{\leq}\pi'(y_{\pi\pi'})$$

$sh(x{\leq}f(y_1,\ldots,y_n))$	$=_{\text{df}}$	$x{=}f(y_1,\ldots,y_n)$
$sh(f(y_1,\ldots,y_n){\leq}x)$	$=_{\text{df}}$	$x{=}f(y_1,\ldots,y_n)$
$sh(x{=}c')$ for $c'{\in}\Sigma$	$=_{\text{df}}$	$x{=}c$
$sh(\varphi_1 \wedge \varphi_2)$	$=_{\text{df}}$	$sh(\varphi_1) \wedge sh(\varphi_2)$

Table 3.6: Defining a subtype constraint $sh(\varphi)$ over $\{f, c\}$ for a subtype constraint φ over Σ.

Since there are only finitely many distinct pairs (x_π, y_π) of variables in $V\varphi$, at least one such pair must be repeated for sufficiently large π and π'. This pumping argument establishes a cycle clash *C4* in φ which contradicts our assumption. Thus, $\mathsf{s}_\varphi(z)$ must be finite for all $z \in V\varphi$.

3.3 Structural subtyping

We show how to reduce structural to non-structural satisfiability. To distinguish between the structural and non structural case, we consider the subtype constraint:

$$x{\leq}f(y) \wedge x{\leq}f(z) \wedge z{\leq}f(y) \wedge y{=}c$$

which is satisfiable over a non-structural signature $\{\bot, \top, f, c\}$, *e.g.* $x{\mapsto}\bot$, but not over any structural signature.

Therefore, we need an auxiliary shape test which verifies that a structural subtype constraint entails at most one unique shape for each of its variables. For this we use a standard technique [Tiu92] to characterize all shapes of solutions by a second, new constructed structural subtype constraint $sh(\varphi)$ defined over a shrunken signature $\{f, c\}$. Let φ be a subtype constraint, the constraint $sh(\varphi)$ is defined as φ where (1) all constants are substituted by c and (2) all inequalities are complicated to be equalities (see Table 3.6).

The origin constraint φ is called *weakly unifiable* (over finite, regular, resp. possibly infinite trees) if and only if $sh(\varphi)$ is satisfiable (over finite, regular, resp. possibly infinite trees). Since $sh(\varphi)$ is a conjunction of equality constraints, its satisfiability can be checked by first-order unification. We refer to Baader and Siekmann [BS94] for an introduction in unification theory. In case of finite trees, weakly unifiability also subsumes the cycle clash which we have defined for non-structural subtype satisfiability.

Proposition 3.5
A structural subtype constraint is satisfiable over finite, regular, resp. possibly infinite trees if and only if it does not contain any label clash and it is weakly unifiable over finite, regular, resp. possibly infinite trees.

Its proof mainly depends on the following lemma which is independently shown by Frey [Fre02] and also by Tiuryn [Tiu92]:

Lemma 3.9
If φ is satisfiable, let α be a solution of $sh(\varphi)$. Then φ has a solution β that is of the same shape as α, i.e., *for all $x \in V(\varphi) = V(sh(\varphi))$, $sh(\alpha(x) = \beta(x))$ is unifiable.*

3.4 Summary and complexity analysis

So far, we have seen that we can decide satisfiability of structural and non-structural subtype constraints by testing for the existence of different kinds of clashes.

Definition 3.1
We call a constraint φ clash free *for the structure of*

1. *possibly infinite trees if it does not contain any label clash,*
2. *regular trees if it does not contain any label clash,*
3. *finite trees if it does neither contain a label clash nor a cycle clash.*

We now consider efficiency issues.

The existence of a label clash in a constraint φ can be tested in cubic time in the size of φ. First, one computes a table of quadratic size that stores all valid judgments $\varphi \vdash u \leq v$. Second, one compares the labels required by φ for all u and v with $\varphi \vdash u \leq v$. The existence of a cycle clash can also be tested in cubic time.

In case of structural constraints we have additionally to check weakly unifiability which corresponds to classical first-order unification. Complete algorithms for first-order unification over finite [Rob65] and regular or infinite trees [MM82] were given and enhanced in its worst case complexity [PW78]. Finally, first-order unification is shown to be linear for finite trees and quasi-linear for regular or infinite trees. Altogether, this yields the following result:

THEOREM 3.1
A subtype constraint φ is satisfiable (over finite, regular, resp. possibly infinite trees) if and only if it is clash-free (over finite, regular, resp. possibly infinite trees) and in the structural case it is additionally weakly unifiable. In all cases, satisfiability can be tested in cubic time in the size of φ. Least and greatest solutions of satisfiable constraints exist for the non-structural infinite and regular cases, but not necessarily over finite trees.

PROOF. From Propositions 3.1, 3.2, 3.5, 3.3, 3.4 and the above discussion on efficiency.

4 Subtyping over a Poset

So far we have investigated subtype satisfiability where constants are ordered in a lattice. In this chapter we generalize subtype satisfiability by constants ordered in an arbitrary poset.

For subtype satisfiability over posets, there exist only partial answers [TW93, Tiu92, Fre02]. The complexity of structural subtype satisfiability is not settled in case of recursive types. Decidability and complexity of non-structural subtype satisfiability are open for both finite and recursive types.

We present a uniform treatment to determine the complexities of all these classes: finite versus recursive types, and structural versus non-structural orderings. It based on a new approach, connecting subtype constraints and modal logic. As an intermediate result we introduce a new subtype language of uniform subtype constraints and show that their satisfiability problem is polynomial time equivalent to that of a dialect of propositional dynamic logic.

4.1 Propositional dynamic logic over trees

Propositional dynamic logic (*PDL*) is a modal logic that extends Boolean logic to directed graphs of possible worlds. The same proposition may hold in some node of the graph and be wrong in others. Nodes are connected by labeled edges, that can be talked about modal operators.

In this paper, we consider the modal logic PDL_n, the *PDL* language for the complete infinite n-ary tree. PDL_n is naturally subsumed by the monadic second-order logic SnS of the complete n-ary tree [Rab69].

4.1.1 Other PDL dialects

Propositional dynamic logic (*PDL*) over directed edge-labeled graphs goes back to Fischer and Ladner [FL79], who restricted Pratt's dynamic logic to the propositional fragment. It is well known that *PDL* has the *tree property*: every satisfiable *PDL*

$$
\begin{array}{lcl}
R & ::= & i \mid R \cup R' \mid RR' \mid R^* \quad \text{where } 1 \leq i \leq n \\
D & ::= & p \mid \neg D \mid D \wedge D' \mid [R]D
\end{array}
$$

Figure 4.1: Syntax of PDL_n.

formula can be satisfied in a rooted edge-labeled tree. Deterministic *PDL* [HKT00, BAHP82, VW86] restricts the model class to graphs whose edge labels are functional in that they determine successor nodes. Deterministic *PDL* with edge labels $\{1, \ldots, n\}$ is the closest relative to our language PDL_n, due to the tree property.

Besides of PDL_n, a large variety of PDL dialects with tree models were proposed in the literature. These differ in the classes of tree models, the permitted modal operators, and the logical connectives. Three different dialects of PDL over finite, binary, or n-ary trees were proposed in [BMV94, Kra97, Pal99], see [BGM03] for a comparison. PDL over finite unranked ordered trees were proposed for computational linguistics applications [BMV94] and found recent interest for querying XML documents.

4.1.2 PDL_n and its fragments

For every $n \geq 1$ we define a logic PDL_n as the PDL logic, for describing the complete infinite n-ary tree.

The syntax of PDL_n expressions[1] A is given in Figure 4.1. Starting from some infinite set *Prop* of propositional variables p, it extends the Boolean logic over these variables by universal modalities $[R]A$, where R is a regular expression over the alphabet $\{1, \ldots, n\}$.

We frequently use the modality $[*]$ as an abbreviation of $[\{1, \ldots, n\}^*]$, and sometimes $[+]$ as a shorthand for $[\{1, \ldots, n\}^+]$. We freely use definable logical connective for implication $\rightarrow$, equivalence $\leftrightarrow$, disjunction $\vee$, exclusive disjunction $\dot{\vee}$, and the Boolean constants *true* and *false*. Furthermore, we can define existential modalities $\langle R \rangle D$ by $\neg[R]\neg D$.

We interpret formulas of PDL_n over the complete infinite n-ary trees. Tree nodes are labeled by the set of propositions that are valid there. Formally, a model M of a formula in PDL_n assigns Boolean values $0, 1$ to propositional variables in every node in $\{1, \ldots, n\}^*$: $M : Prop \times \{1, \ldots, n\}^* \rightarrow \{0, 1\}$. Table 4.1 defines when a formula D holds in some node π of some model M, in formulas: $M, \pi \models D$. A formula $[R]D$ is

[1] We could allow for test $?A$ in regular expressions, which frequently occur in PDL dialects but we will not need them.

$M, \pi \models p$	if $M(p, \pi) = 1$
$M, \pi \models D_1 \wedge D_2$	if $M, \pi \models D_1$ and $M, \pi \models D_2$
$M, \pi \models \neg D$	if not $M, \pi \models D$
$M, \pi \models [R]D$	if for all $\pi' \in L(R)$: $M, \pi\pi' \models D$

Table 4.1: Semantics of PDL_n.

valid for some node π of a tree M if D holds in all R descendants of π in M, *i.e.*, in all nodes $\pi\pi'$ where π' belongs to the language $L(R)$ of R.

Let us recall some logical notations. A formula D is *valid in a model* M if it holds in the root of M: $M \models D$ iff $M, \varepsilon \models D$. A formula D is *satisfiable* if it is valid in some model; it is *valid* it is valid in all models: $\models D$ iff $\forall M.\, M \models D$. Two formulas D, D' are equivalent if $D \leftrightarrow D'$ is valid: $D \models\!\models D'$ iff $\models D \leftrightarrow D'$. For instance, $\langle i \rangle A \models\!\models [i]A$ holds for all $1 \leq i \leq n$ and all A, since nodes of the n-ary tree have unique i successors.

Note that PDL_n respects the substitution property: whenever $D_1 \models\!\models D_2$ then $D[D_1/D_2] \models\!\models D$. To see this note that if $D_1 \models\!\models D_2$ then the equivalence $D \leftrightarrow D'$ is valid not only at the root of all models but also at all other nodes of all models. This is because all subtrees of complete n-ary trees are again complete n-ary trees.

THEOREM 4.1
Satisfiability of PDL_n formulas is in DEXPTIME.

A PDL_n formula is satisfiable iff it can be satisfied by a deterministic rooted graph with edge labels in $\{1, \ldots, n\}$. The proposition thus follows from the *DEXPTIME* upper bound for deterministic *PDL* [HKT00, BAHP82], which is a corollary to the analogous result for *PDL*.

4.1.3 Flat core PDL_n

We next investigate lower complexity bounds for PDL_n. It is known from Vardi and Wolper [VW86] that satisfiability of deterministic *PDL* is *DEXPTIME*-complete. This result clearly carries over to PDL_n.

An analysis of Spaan's proofs [Spa93] reveals that nested $[*]$ modalities are not needed for *DEXPTIME*-hardness. But we can even do better, *i.e.*, restrict the language further.

$$
\begin{array}{lcll}
B & ::= & p_1 \wedge p_2 \mid \neg p \mid [i]p & \text{where } 1 \leq i \leq n \\
C & ::= & p \mid [*]\,(p \leftrightarrow B) \mid C_1 \wedge C_2 &
\end{array}
$$

Figure 4.2: Syntax of flat core PDL_n.

We define the fragment *flat core* PDL_n in Figure 4.2. A formula of flat core PDL_n is a conjunction of propositional variables and expressions of the form $[*]\,(p \leftrightarrow B)$. Note that $[*]$ modalities cannot be nested. Furthermore, all Boolean sub-formulas B are flat in that Boolean connectives only apply to variables.

THEOREM 4.2
Satisfiability of flat core PDL_n formulas is DEXPTIME-complete.

We prove that satisfiability of flat core PDL_n is *DEXPTIME*-hard and thus *DEXPTIME*-complete (Theorem 4.1).

We proceeds in two steps. We first introduce a new dialect of PDL_n that we call the *core of* PDL_n , and express emptiness of intersections of tree automata in that language. This proofs *DEXPTIME* hardness [Sei94] of core PDL_n. In the second step, we normalize core PDL_n into flat core PDL_n. This relies on a flattening procedure inspired by techniques of Spaan [Spa93].

4.1.4 Core PDL_n

The core of PDL_n is a fragment of PDL_n that is slightly richer than flat core PDL_n. Formulas C of core PDL_n are conjunctions of propositional variables and expression $[*]B$, where B is an arbitrary, possibly non-flat Boolean expression.

$$
\begin{array}{lcl}
B & ::= & p \mid \neg B \mid B \wedge B' \mid B \leftrightarrow B' \mid [i]B \\
C & ::= & B \mid [*]B \mid C \wedge C
\end{array}
$$

The modalities are again restricted to immediate $[i]$ successors (where $1 \leq i \leq n$) and arbitrary $[*]$ descendants such that $[*]$ cannot be nested below other modalities.

In Table 4.2 we define a set of standard operators on Boolean expressions in core PDL_n formulas. Note that all operators affect the size of formulas linearly. Our syntax provides for equivalences, to avoid the exponential blow up in the standard:

$$D_1 \leftrightarrow D_2 \models\mid D_1 \rightarrow D_2 \wedge D_2 \rightarrow D_1$$

But expressing nested equivalences through two-sided implications might blow up sizes exponentially.

or.	$B_1 \vee B_2$	$=_{\text{df}} \neg(\neg B_1 \wedge \neg B_2)$
implication.	$B_1 \rightarrow B_2$	$=_{\text{df}} \neg B_1 \vee B_2$
exclusive or.	$B_1 \dot{\vee} B_2$	$=_{\text{df}} \neg(B_1 \leftrightarrow B_2)$
false value.	*false*	$=_{\text{df}} p_0 \wedge \neg p_0$ for some $p_0 \in Prop$
true value.	*true*	$=_{\text{df}} \neg$*false*

Table 4.2: Operators on Boolean expressions.

Proposition 4.1
Satisfiability of core PDL_n formulas is DEXPTIME-hard.

PROOF. This can be proved by a closer inspection of *DEXPTIME*-hardness proofs for *PDL* [Spa93, HKT00, BdRV01] or deterministic *PDL* [VW86]. Here, we give a new direct proof by encoding emptiness of intersections of tree automata.

Let Σ be a finite ranked signature. A *tree automaton* A over a signature Σ consists of a finite set $states(A)$ of *states*, a subset $final(A) \subseteq states(A)$ of *final states*, and a set $rules(A)$ of *transition rules* of the form $f(q_1, \ldots, q_n) \rightarrow q$ where $q_1, \ldots, q_n, q \in states(A_i)$ and $n = arity_\Sigma(f)$. The language of a tree automaton $L(A)$ contains all those ground terms over Σ that can be evaluated into a final state by rule application.

We first encode trees over Σ in PDL_n with max successors where max is the maximal arity of function symbols in Σ. We introduce fresh propositional variables p_f for every symbol $f \in \Sigma$ to represent f-labeled nodes, and a propositional variable p_{dom} to express tree domains. A model M encodes a tree τ if for all $\pi \in \{1, \ldots, max\}^*$:

$$\begin{array}{lcll} M, \pi & \models & p_{dom} & \text{iff} \quad \pi \in dom(\tau) \qquad \text{and} \\ M, \pi & \models & p_f & \text{iff} \quad \tau(\pi) = f \end{array}$$

Lemma 4.1
There exists a formula $tree_\Sigma$ in the core of PDL_n whose models represent precisely the trees in $tree_\Sigma$.

PROOF. We use a couple of well-formedness conditions for representations of possibly infinite trees. Formula $label_\Sigma$ says that the root of every ground term belongs to its domain and every node of the domain is labeled in Σ.

$$label_\Sigma =_{\text{df}} p_{dom} \wedge [*](p_{dom} \rightarrow \dot{\bigvee_{f \in \Sigma}} p_f)$$

Condition $arity_\Sigma$ requires that every node of a tree fulfills the arity required by its label.

$$arity_\Sigma =_{\text{df}} [*](\bigwedge_{f\in\Sigma} p_f \rightarrow \begin{array}{l}(\bigwedge_{i=1}^{arity_\Sigma(f)}[i]p_{dom}\wedge \\ \bigwedge_{j=1+arity_\Sigma(f)}^{max}[j]\neg p_{dom}))\end{array}$$

Property *prefix* restricts tree domains of ground terms to be prefixed-closed:

$$prefix =_{\text{df}} [*]p_{dom} \rightarrow \left(\bigwedge_{i=1}^{max}[i]\ p_{dom}\right)$$

Possibly infinite trees are now definable:

$$tree_\Sigma =_{\text{df}} label_\Sigma \wedge arity_\Sigma \wedge prefix$$

We next want to restrict models to representation ground terms, i.e., to finite trees over Σ, but unfortunately, finiteness cannot be expressed in PDL_n. Lemma 4.2 indicates a way out of this problem. It is sufficient to restrict the depth of terms exponentially, rather then to impose finiteness.

Lemma 4.2
Let $(A_i)_{i=1}^n$ be a finite sequence of tree automata over the same signature. If the intersection $\cap_{i=1}^n L(A_i)$ is nonempty, then it contains some tree of depth bounded by $\prod_{i=1}^n |states(A_i)|$.

PROOF. We can construct a tree automaton for the intersection with at most $\prod_{i=1}^n |states(A_i)|$ and then apply the pumping lemma for regular tree languages.

It is thus sufficient to encode ground terms whose depth is bounded exponentially in the size of the given intersection of tree automata. This can be expressed by a PDL_n formula of polynomial size, which simulates a counter.

Lemma 4.3
For every $n \geq 0$, there exists a formula $ground{-}term_\Sigma(n)$ in the core of PDL_n describing all finite trees over Σ with depth bounded by 2^n.

PROOF. Condition $counter(n)$ describes an n-bit counter that counts the depth of nodes starting from the root. We consider tree models with propositional variables $(p_i)_{i=1}^n$ that represent the n bits of the counter. Lets identify the Boolean values t with the digit 1 and f with 0. For every model M and node π the sequence

$M(p_n, \pi) \ldots M(p_1, \pi)$ is the binary representation of the depth of node π in tree M, modulo 2^n.

$$\begin{array}{rcl} \mathit{all}(n) & =_{\mathrm{df}} & \bigwedge_{i=1}^{n} p_i \\ \mathit{counter}(0) & =_{\mathrm{df}} & \mathit{true} \\ \mathit{counter}(n) & =_{\mathrm{df}} & \mathit{counter}(n{-}1) \wedge \neg p_n \\ & & \wedge[*]((\neg p_n \wedge \neg \mathit{all}(n{-}1)) \rightarrow \wedge_{i=1}^{n}[i]\neg p_n) \\ & & \wedge[*]((\neg p_n \wedge \mathit{all}(n{-}1)) \rightarrow \wedge_{i=1}^{n}[i] p_n) \\ & & \wedge[*]((p_n \wedge \neg \mathit{all}(n{-}1)) \rightarrow \wedge_{i=1}^{n}[i] p_n) \\ & & \wedge[*]((p_n \wedge \mathit{all}(n{-}1)) \rightarrow \wedge_{i=1}^{n}[i]\neg p_n) \end{array}$$

The formula $\mathit{depth}(n)$ bounds the depth of nodes in the domain to 2^n.

$$\mathit{depth}(n) =_{\mathrm{df}} \mathit{counter}(n) \wedge [*](\mathit{all}(n) \rightarrow \neg p_{\mathit{dom}})$$

We can now define ground terms:

$$\mathit{ground{-}term}_{\Sigma}(n) =_{\mathrm{df}} \mathit{tree}_{\Sigma} \wedge \mathit{depth}(n)$$

Let $(A_i)_{i=1}^{n}$ be a sequence of tree automata over a signature Σ with disjunct state sets. We encode simultaneously accepting runs of all tree automata $(A_i)_{i=1}^{n}$. We use propositional variables p_q for all states $q \in \uplus_{i=1}^{n} \mathit{states}(A_i)$.

$$\begin{array}{rcl} \mathit{run}(A_i) & =_{\mathrm{df}} & [*]\ (\wedge_{q \in \mathit{states}(A)}\ ((p_q \wedge p_f) \rightarrow \\ & & \quad \vee_{\ f(q_1,\ldots,q_n) \rightarrow q \in \mathit{rules}(A_i)} \wedge_{i=1}^{n}\ [i] p_{q_i})) \\ \mathit{accept}(A_i) & =_{\mathrm{df}} & \vee_{q \in \mathit{final}(A_i)} p_q \wedge \mathit{run}(A_i) \end{array}$$

Lemma 4.4
The intersection $\cap_{i=1}^{n} L(A_i)$ is the set of ground terms that yield models of the following PDL_n formula for $k = \max_{i=1}^{n} |\mathit{states}(A_i)|$:

$$\mathit{ground{-}term}(\lceil \log(k) \rceil * n) \wedge \wedge_{i=1}^{n} \mathit{accept}(A_i)$$

4.1.5 Flattening

Proposition 4.2
Every core PDL_n formula C is satisfaction equivalent to some flat core PDL_n formula, that can be computed in linear time.

$$
\begin{array}{rcl}
\mathit{flat}(p) & =_{\mathrm{df}} & [*]\,(P_p \leftrightarrow p \wedge p) \\
\mathit{flat}(\neg B) & =_{\mathrm{df}} & [*]\,(P_{\neg B} \leftrightarrow \neg P_B) \wedge \mathit{flat}(B) \\
\mathit{flat}(B \wedge B') & =_{\mathrm{df}} & [*]\,(P_{B \wedge B'} \leftrightarrow (P_B \wedge P_{B'})) \\
 & & \wedge\, \mathit{flat}(B) \wedge \mathit{flat}(B') \\
\mathit{flat}([i]B) & =_{\mathrm{df}} & [*]\,(P_{[i]B} \leftrightarrow [i]P_B) \wedge \mathit{flat}(B) \\
\mathit{flat}(B \leftrightarrow B') & =_{\mathrm{df}} & [*]\,(P_{B \leftrightarrow B'} \leftrightarrow (P_{B \rightarrow B'} \wedge P_{B' \rightarrow B})) \\
 & & \wedge\, \mathit{flat}(B \rightarrow B') \wedge \mathit{flat}(B' \rightarrow B) \\
\mathit{flat}_2(B) & =_{\mathrm{df}} & P_B \wedge \mathit{flat}(B) \\
\mathit{flat}_2([*]B) & =_{\mathrm{df}} & [*]P_B \wedge \mathit{flat}(B) \\
\mathit{flat}_2(C \wedge C) & =_{\mathrm{df}} & \mathit{flat}_2(C) \wedge \mathit{flat}_2(C')
\end{array}
$$

Table 4.3: Flattening core PDL_n formulas.

The idea of the proof is to introduce new propositional variables for all sub-term positions of a given PDL_n formula. We fix a finite set $Prop_0$ of propositional variables and an injective generator function:

$$P : PDL_n \rightarrow (Prop - Prop_0)$$

that maps a PDL_n formulas D to propositional variables P_D. Given this generator, Table 4.3 defines two flattening functions flat and flat_2, for core PDL_n formulas of type B and C respectively.

Formulas $\mathit{flat}(B)$ and $\mathit{flat}_2(C)$ are clearly flat core PDL_n formulas for all core PDL_n formulas B and $\mathit{flat}_2(C)$, except for sub-formulas $[*]P_B$ which can be expressed through $[*](P_B \leftrightarrow \mathit{true})$ and thus by the flat formula: $[*](P_B \leftrightarrow P_{\mathit{true}} \wedge P_{\mathit{true}}) \wedge \mathit{flat}(\mathit{true})$.

The sizes of $\mathit{flat}(B)$ and $\mathit{flat}_2(C)$ remain linear in those of B and C respectively, when sharing common subconstraints $\mathit{flat}(B_1)$ and $\mathit{flat}(B_2)$ in translations of equivalences $\mathit{flat}(B_1 \leftrightarrow B_2)$, i.e., in $\mathit{flat}(B_1 \rightarrow B_2)$ and $\mathit{flat}(B_2 \rightarrow B_1)$.

Lemma 4.5 (Correctness)
For all core PDL_n formulas C with $Prop(C) \subseteq Prop_0$:

$$C \models\mid \exists Prop{-}Prop_0.\ \mathit{flat}_2(C)$$

The proof relies on the following two Lemmas.

$$
\begin{array}{rcll}
M'(p,\pi) & =_{\text{df}} & M(p,\pi) & \text{if } p \in \mathit{Prop}_0 \\
M'(p,\pi) & =_{\text{df}} & \text{arbitrary} & \text{if } p \notin \mathit{Prop}_0 \cup P(B) \\
M'(P_p,\pi) & =_{\text{df}} & M'(p,\pi) & \\
M'(P_{\neg B},\pi) & =_{\text{df}} & \neg M'(B,\pi) & \\
M'(P_{B_1 \wedge B_2},\pi) & =_{\text{df}} & M'(B_1,\pi) \wedge M'(B_2,\pi) & \\
M'(P_{B_1 \leftrightarrow B_2},\pi) & =_{\text{df}} & M'(B_1,\pi) \leftrightarrow M'(B_2,\pi) & \\
M'(P_{[i]B},\pi) & =_{\text{df}} & \forall\, 1 \leq i \leq n. M'(B,\pi i) &
\end{array}
$$

Figure 4.3: Extending models to new variables P_B.

Lemma 4.6
For all core PDL_n formulas B with variables $\mathit{Prop}(B) \subseteq \mathit{Prop}_0$:

$$\models \forall \mathit{Prop}_0\ \exists \mathit{Prop} - \mathit{Prop}_0.\ \mathit{flat}(B)$$

PROOF. We fix a model $M : \mathit{Prop} \times \{1, \ldots n\}^* \to \{0,1\}$ of B and define a model $M' : \mathit{Prop} \times \{1, \ldots n\}^* \to \{0,1\}$ of $\mathit{flat}(B)$ in Figure 4.3. The definition of $M'(P_B, \pi)$ is by induction on the structure of terms B. Clearly, M' differs from M only on variables in $\mathit{Prop} - \mathit{Prop}_0$. We can show $M' \models \mathit{flat}(B)$ for all formulas B with variables $\mathit{Prop}(B) \subseteq \mathit{Prop}_0$ by induction on the structure of B.

Lemma 4.7
$\mathit{flat}(B) \models [*]\,(P_B \leftrightarrow B)$ for all core PDL_n formulas B.

PROOF. By induction on the structure of formulas B.

1. Let $B = p$ then $\mathit{flat}(p) \models [*]\,(P_p \leftrightarrow p)$ (Table 4.3).
2. Let $B = B_1 \wedge B_2$ (the remaining cases $B = \neg B'$ or $B = B_1 \leftrightarrow B_2$ are analogous). It holds that $\mathit{flat}(B) \models [*]\,(P_B \leftrightarrow (P_{B_1} \wedge P_{B_2})$ (Table 4.3). It further holds for $i \in \{1,2\}$ that $\mathit{flat}(B) \models [*]\,(P_{B_i} \leftrightarrow B_i)$ by induction on B_i. We conclude $\mathit{flat}(B) \models [*]\,(P_B \leftrightarrow B_1 \wedge B_2)$.
3. Case $B = [i]\,B'$. It hold that $\mathit{flat}(B) \models [*]\,(P_B \leftrightarrow [i]P_{B'})$ (Table 4.3) and $\mathit{flat}(B) \models [*]\,(P_{B'} \leftrightarrow B')$ by induction on B'. It follows that $\mathit{flat}(B) \models [*][i]\,(P_{B'} \leftrightarrow B')$ also holds. Again we conclude $\mathit{flat}(B) \models [*]\,(P_B \leftrightarrow [i]B')$.

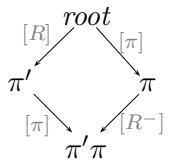
PROOF. [of Correctness Lemma 4.5] by induction on C. Let $C = [*]B$ (the case $C = B$ will be subsumed). To prove is $[*]B \models\!\mid \exists \mathit{Prop} - \mathit{Prop}_0\ ([*]P_B \wedge \mathit{flat}(B))$ for all core PDL_n formulas B with $\mathit{Prop}(B) \subseteq \mathit{Prop}_0$ (see Table 4.3). Lemma 4.7 yields $\mathit{flat}(B) \models [*]\,(P_B \leftrightarrow B)$ and thus $[*]\,P_B \wedge \mathit{flat}(B) \models [*]\,B$. Since $\mathit{Prop}(B) \subseteq \mathit{Prop}_0$

this is equivalent to $\exists Prop{-}Prop_0.[*]\, P_B \wedge \mathit{flat}(B) \models [*]\, B$. The converse follows from Lemma 4.6:

$$\begin{array}{rcl} [*]\, B & \models\!\models & [*]\, B \wedge \forall Prop_0.\ \exists Prop{-}Prop_0.\ \mathit{flat}(B) \\ & \models & [*]\, B \wedge \exists Prop{-}Prop_0.\ \mathit{flat}(B) \\ & \models & \exists Prop{-}Prop_0\ ([*]\, B \wedge \mathit{flat}(B)) \end{array}$$

4.1.6 Inverted PDL_n

The modal operator $[R]$ of PDL_n addresses all R descendants of the actual world. We now consider a variant of PDL_n with inverted modalities $[R]^-$, which address all nodes $\pi'\pi$ reached by prefixing some $\pi' \in L(R)$ to the actual node π:



Thus, $[R][R']^-$ equals to $[R'][R]$. We define:

$$M, \pi \models [R^-]D \ \text{ if for all } \pi' \in L(R)\text{: } M, \pi'\pi \models D.$$

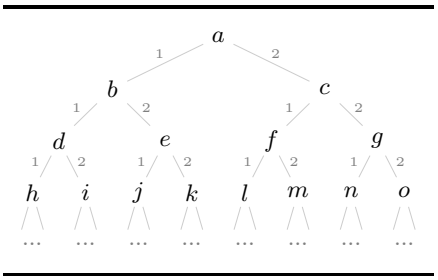
Inverted flat core PDL_n is defined in analogy to flat core PDL_n except that all modalities are inverted.

$$\begin{array}{rcll} B & ::= & p_1 \wedge p_2 \mid \neg p \mid [i]^- p & \text{for } 1 \leq i \leq n \\ C & ::= & p \mid [*]\,(p \leftrightarrow B) \mid C_1 \wedge C_2 & \end{array}$$

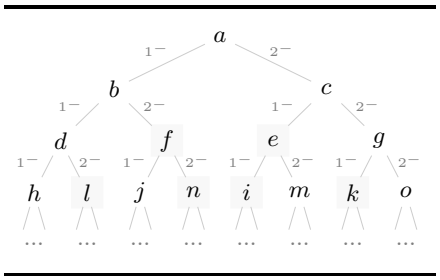
We will freely omit inversion for $[*]$ operators, as these are never nested below modalities. We can translate flat core PDL_n formulas C into formulas C^- of the inverted flat core, and vice versa, by replacing the operators $[i]$ through $[i]^-$. Models can be inverted too: $M^-(p, \pi) = M(p, \pi^{-1})$ where π^{-1} is the inversion of π.

Lemma 4.8
$M \models C$ iff $M^- \models C^-$.

The following figures compare a model with its inversion (all nodes changing its position due to path inversion are grey underlined):



5.1. A tree model ...



5.2. compared with its inversion.

$$
\begin{array}{rcl}
[\![x{=}f(x_1,\ldots,x_n)]\!] & =_{\mathrm{df}} & P_{x=f} \;\wedge\; \bigwedge_{g\in\Sigma}\bigwedge_{1\le i\le n}[*]\,(P_{x_i=g} \leftrightarrow [i]^- P_{x=g}) \\
[\![x{\le}y]\!] & =_{\mathrm{df}} & [*]\bigvee_{f\le_\Sigma g}(P_{x=f}\wedge P_{y=g}) \\
[\![\varphi_1\wedge\varphi_2]\!] & =_{\mathrm{df}} & [\![\varphi_1]\!]\wedge[\![\varphi_2]\!]
\end{array}
$$

Table 4.4: Expressing uniform covariant subtype constraints in inverted PDL_n.

4.2 Uniform subtype satisfiability

In this section, we investigate the complexity of uniform subtype satisfiability. We first show how to encode uniform subtype constraints into inverted PDL_n. We then give a translation from *inverted flat core* PDL_n back to uniform subtype satisfiability. Both translations are polynomial time and preserve satisfiability (Proposition 4.4 and 9.1). The complexity of PDL_n (Theorem 4.2) thus carries over to uniform subtype satisfiability.

THEOREM 4.3
Uniform subtype satisfiability over possibly infinite trees is DEXPTIME-complete.

4.2.1 Uniform subtype constraints into PDL_n

We encode uniform subtype constraints over infinite trees into inverted PDL_n. The translation relies on ideas of Tiuryn and Wand [TW93], but it is simpler with modal logics as the target language, rather than infinite sets of regular path constraints. We first present our translation for covariant uniform signatures and then sketch the contravariant case.

Let Σ be a uniform covariant signature and $n > 1$ the arity of its function symbols. We fix a finite set of type variables V and consider subtype constraints φ over Σ with

$V(\varphi) \subseteq V$. For all $x \in V$ and $f \in \Sigma$ we introduce propositional variables $P_{x=f}$ that are true at all nodes $\pi \in \{1,\dots,n\}^*$ where the label of x is f.

The *well-formedness formula* wff_V states that all nodes of tree values of all $x \in V$ carry a unique label f:

$$\mathit{wff}_V \;=_{\mathrm{df}}\; \bigwedge_{x\in V} [*]\,(\dot{\bigvee}_{f\in\Sigma} P_{x=f}).$$

A polynomial time encoding of subtype constraints is presented in Table 4.4. Inverted modalities $[i]^-$ are needed to translate $x{=}f(x_1,\dots,x_n)$ since $\alpha \models x{=}f(x_1,\dots,x_n)$ if and only if $\alpha(x)(\varepsilon) = f$ and $\alpha(x)(i\pi) = \alpha(x_i)(\pi)$ for all words $i\pi \in \{1,\dots,n\}^*$.

Proposition 4.3
A uniform subtype constraint φ over a covariant signature Σ with $V(\varphi) \subseteq V$ is satisfiable if and only if $\mathit{wff}_V \wedge [\![\varphi]\!]$ is satisfiable.

PROOF. A solution of φ is a function $\alpha : V \to \mathit{tree}_\Sigma$. Let n be the arity of function symbols in Σ, so that all trees in tree_Σ are complete n-ary trees with nodes labeled in Σ, i.e., total functions of type $\{1,\dots,n\}^* \to \Sigma$. A variable assignment α thus defines a PDL_n model $M_\alpha : \mathit{Prop} \times \{1,\dots,n\}^* \to \Sigma$ that satisfies for all $x \in V$ and $\pi \in \{1,\dots,n\}^*$:

$$M_\alpha(P_{x=f}, \pi) \;\leftrightarrow\; \alpha(x)(\pi) = f.$$

We can now show by induction on the structure of φ that $\alpha \models \varphi$ iff $M_\alpha, \varepsilon \models \mathit{wff}_V \wedge [\![\varphi]\!]$.

Proposition 4.4
Uniform subtype satisfiability with covariant signatures over possibly infinite trees is in DEXPTIME.

PROOF. It remains to show that our reduction is in polynomial time. This might seem obvious, but it needs some care. Exclusive disjunctions of the form $p_1 \dot{\vee} \dots \dot{\vee} p_n$ as used in the well-formedness formula can be encoded in quadratic time through $\bigvee_{i=1}^n (p_i \wedge \bigwedge_{1\le j\ne i\le n} \neg p_j)$. Equivalences $p \leftrightarrow \neg p'$ as used can be encoded in linear time by $(p \wedge \neg p') \vee (\neg p \wedge p')$.

Contravariance. Our approach smoothly extends to uniform subtyping with contravariant signatures. The key idea is that we can express polarities in *inverted* flat core PDL_n by using a new propositional variable p_{pol}. For example, consider the uniform signature $\Sigma = \{\to\}$, where $\to$ is the usual function type constructor. The variable p_{pol} is true in nodes with polarity 1 and false otherwise:

$$p_{pol} \wedge [*]\,(p_{pol} \leftrightarrow [1]^- \neg p_{pol}) \;\wedge\; [*]\,(p_{pol} \leftrightarrow [2]^- p_{pol}).$$

Limitation due to Inversion. Inversion is crucial to our translation and has a number of consequences. Most importantly, we cannot express the formula $[*](p \to [+]p')$ in inverted PDL_n, which states that whenever p holds at some node then p' holds for in all proper descendants.

As a consequence, we cannot directly translate subtype constraints over standard signatures into PDL_n (which we consider in Sections 4.3). The difficulty is to encode tree domains in the presence of leafs. Suppose we want to define that p holds for all nodes outside the tree domain. We could do so by imposing $[*](p_c \to [+]p)$ for all constants c, but this is impossible in inverted PDL_n.

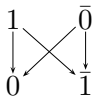
This is not a problem for uniform signatures, because every tree is complete n-ary. Thus we do not need to express tree domains when considering satisfiability. Unfortunately, the same technique does not extend to entailment and other first-order fragments that require negations.

4.2.2 Back translation

To prove *DEXPTIME*-hardness of uniform subtype satisfiability, we show how to express inverted flat core PDL_n by uniform subtype constraints, indeed only with covariant signatures. Our encoding of Boolean logic is inspired by Tiuryn [Tiu92], while the idea to lift this encoding to PDL_n is new.

Let C be a formula of inverted flat core PDL_n. We aim to find a subtype constraints $[\![C]\!]^{-1}$ which preserves satisfiability. The critical point is how to translate PDL_n's negation since it is absent in our target language of uniform subtype constraints.

We work around by constructing a uniform subtype constraints with function symbols ordered in a crown: $\Sigma(n) = \{0, \bar{0}, 1, \bar{1}\}$.



All function symbols have arity n and satisfy $x \leq_{\Sigma(n)} y$ for all $x \in \{0, \bar{1}\}$, $y \in \{1, \bar{0}\}$. The symbols 0 and 1 model PDL_n's underlying boolean lattice $bool = \{0, 1\}$; the additional two symbols are introduced to define negation by $diag(c) = \overline{c}$ for $c \in bool$.

Next, Table 4.5 shows how to define *diag* by a subtype constraint. For every propositional variable p we introduce a new type variables X_p in the subtype constraint we are constructing to. The subtype constraint *all-c*(x) holds for the unique trees that is completely labeled by some $c \in \Sigma(n)$. The subtype constraint *all-bool*(x) holds for

$\mathit{all\text{-}c}(x)$	$=_{\mathrm{df}}$	$x{=}c(x,\ldots,x)$ for some $c \in \Sigma(n)$
$\mathit{all\text{-}bool}(x)$	$=_{\mathrm{df}}$	$\exists y \exists z.\ \mathit{all}\text{-}0(x) \wedge x{\leq}y{\leq}z \wedge \mathit{all}\text{-}1(z)$
$\mathit{all\text{-}}\overline{\mathit{bool}}(x)$	$=_{\mathrm{df}}$	$\exists y \exists z.\ \mathit{all}\text{-}\bar{1}(x) \wedge x{\leq}y{\leq}z \wedge \mathit{all}\text{-}\bar{0}(z)$
$\mathit{upper}(x,y)$	$=_{\mathrm{df}}$	$\exists z.\ x{\leq}z \wedge y{\leq}z$
$\mathit{lower}(x,y)$	$=_{\mathrm{df}}$	$\exists z.\ z{\leq}x \wedge z{\leq}y$
$\mathit{diag}(x,y)$	$=_{\mathrm{df}}$	$\mathit{all\text{-}bool}(x) \wedge \mathit{all\text{-}}\overline{\mathit{bool}}(y) \wedge \mathit{upper}(x,y) \wedge \mathit{lower}(x,y)$
$\mathit{all}(p_1 \vee p_2 \vee \neg p_3 \vee \neg p_4)$	$=_{\mathrm{df}}$	$\exists z.\ \bigwedge_{1\leq i\leq 4}\ \mathit{all\text{-}bool}(X_{p_i})$
		$\wedge \mathit{upper}(z, X_{p_4}) \wedge \mathit{upper}(z, \mathit{diag}(X_{p_2}))$
		$\wedge \mathit{lower}(z, X_{p_1}) \wedge \mathit{lower}(z, \mathit{diag}(X_{p_3}))$
$\mathit{all}(p_1 \vee p_2)$	$=_{\mathrm{df}}$	$\exists X_q.\ \mathit{all}(p_1 \vee p_2 \vee \neg q \vee \neg q) \wedge \mathit{all}\text{-}1(X_q)$

Table 4.5: Boolean operations expressed by subtype constraints.

trees that are labeled in *bool*. The constraints $\mathit{lower}(x,y)$ and $\mathit{upper}(x,y)$ require the existence of lower and upper bounds respectively for trees x and y. These bounds are used to define the diagonal pairs $\mathit{diag}(x,y)$ in the crown.

Lemma 4.9
$\mathit{diag}(x,y) \models\!\mid \forall\pi.\ (x(\pi) = 0 \wedge y(\pi) = \bar{0}) \vee (x(\pi) = 1 \wedge y(\pi) = \bar{1})$.

PROOF. Since x is a tree labeled in *bool*, all nodes π satisfy $\alpha(x)(\pi){=}0$ or $\alpha(x)(\pi){=}1$. In the first case (the second is analogous) the constraint $\mathit{lower}(x,y)$ entails $\alpha(y)(\pi){\neq}\bar{1}$. Since y is a $\overline{\mathit{bool}}$ tree, $\alpha(y)(\pi){=}\bar{0}$.

The subtype constraint $\mathit{all}(p_1 \vee p_2 \vee \neg p_3 \vee \neg p_4)$ expresses a universally valid Boolean clause. In its definition, we use the diagonal operator in functional syntax $\mathit{diag}(\mathit{bool}) \to \overline{\mathit{bool}}$ to increase readability. Solutions of such subtype constraints are variable assignments

$$\alpha : \mathit{Prop} \to \{1,\ldots,n\}^* \to \Sigma(n).$$

For variable assignments α into trees over Booleans, we define PDL_n-models $M_\alpha : \mathit{Prop} \times \{1,\ldots,n\}^* \to \mathit{bool}$ by Currying:

$$M_\alpha(p,\pi) = \alpha(X_p)(\pi).$$

Lemma 4.10
Let D be the Boolean formula $p_1 \vee p_2 \vee \neg p_3 \vee \neg p_4$. For all variable assignments α to trees over $\Sigma(n)$, $\alpha \models \mathit{all}(D)$ if and only if M_α is defined and $M_\alpha \models [*]D$.

$$
\begin{array}{rcl}
[\![p]\!]^{-1} & =_{\mathrm{df}} & \exists x_1 \ldots \exists x_m.\ \mathit{all\text{-}bool}(X_p) \wedge X_p{=}1(x_1, \ldots, x_m) \\
[\![[*]\,(p \leftrightarrow [i]^- q)]\!]^{-1} & =_{\mathrm{df}} & \mathit{all\text{-}bool}(X_p) \wedge \mathit{all\text{-}bool}(X_q) \\
 & & \wedge\ \exists x_1 \ldots \exists x_m.(0(x_1, ..., x_m){\leq}X_q{\leq}1(x_1, ..., x_m) \wedge X_p{=}x_i) \\
[\![[*]\,(p \leftrightarrow \neg q)]\!]^{-1} & =_{\mathrm{df}} & \mathit{all}(p \vee q) \wedge \mathit{all}(\neg p \vee \neg q) \\
[\![[*]\,(p \leftrightarrow (q_1 \wedge q_2))]\!]^{-1} & =_{\mathrm{df}} & \mathit{all}(\neg p \vee q_1) \wedge \mathit{all}(\neg p \vee q_2) \wedge \mathit{all}(p \vee \neg q_1 \vee \neg q_2) \\
[\![C_1 \wedge C_2]\!]^{-1} & =_{\mathrm{df}} & [\![C_1]\!]^{-1} \wedge [\![C_2]\!]^{-1}
\end{array}
$$

Table 4.6: Inverted core flat PDL_n in subtype constraints.

PROOF. We assume $\alpha \models \mathit{all}(D)$. The model M_α exists since $\mathit{all}(D) \models \bigwedge_{1 \leq i \leq 4} \mathit{all\text{-}bool}(X_{p_i})$. We show $M_\alpha \models [*]D$ by contradicting the existence of a path π with:

$$\alpha \models X_{p_1}(\pi){=}0 \wedge X_{p_2}(\pi){=}0 \wedge X_{p_3}(\pi){=}1 \wedge X_{p_4}(\pi){=}1.$$

Each of these four disjuncts forbids one of four possible values for $\alpha(z)(\pi)$, as we argue below, where z is the existentially quantified variable in $\mathit{all}(D)$. This is clearly impossible.

1. if $\alpha(X_{p_1})(\pi){=}0$ then $\alpha(z)(\pi) \neq \bar{1}$ since $\alpha \models \mathit{lower}(z, X_{p_1})$;
2. if $\alpha(X_{p_2})(\pi){=}0$ then $\alpha \models \mathit{diag}(X_{p_2})(\pi){=}\bar{0}$ (Lemma 4.9). This implies $\alpha(z)(\pi) \neq 1$ since $\alpha \models \mathit{upper}(z, \mathit{diag}(X_{p_2}))$.
3. if $\alpha(X_{p_3})(\pi){=}1$ then $\alpha \models \mathit{diag}(Xp_3)(\pi){=}\bar{1}$ (Lemma 4.9). This implies $\alpha(z)(\pi) \neq 0$ since $\alpha \models \mathit{lower}(z, \mathit{diag}(X_{p_3}))$.
4. if $\alpha(X_{p_4})(\pi){=}1$ then $\alpha(z)(\pi) \neq \bar{0}$ since $\alpha \models \mathit{upper}(z, X_{p_4})$.

The back translation $[\![C]\!]^{-1}$ of inverted flat core PDL_n into subtype constraints is shown in Table 4.6. All Boolean formulas used there can be expressed by $p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4$ which we have shown how to encode.

Proposition 4.5
Let C be a flat core inverted PDL_n formula. For all variable assignments α to trees over $\Sigma(n)$, $\alpha \models [\![C]\!]^{-1}$ if and only if M_α is defined and $M_\alpha \models C$.

For $n = 0$, subtype constraints become ordering constraints for some given ordering, while PDL_0 satisfiability becomes a Boolean satisfiability problem that is well-known to be NP-complete. We thus obtain a new NP-completeness proof for ordering constraints interpreted over posets [PT96].

4.3 Equivalence of subtype problems

We next show the equivalence of uniform subtype satisfiability with structural and non-structural subtype satisfiabilities over possibly infinite trees. Subtype satisfiability over finite trees will be treated in Section 4.4.

THEOREM 4.4
Structural, non-structural, and uniform subtype satisfiability over possibly infinite trees are equivalent and DEXPTIME-complete.

The proof relies on so called 1-subtype orders which are subtype orders over signatures with a single non-constant, and the corresponding constraints.

1-subtype satisfiability is the satisfiability problem of subtype constraints over 1-subtype orders. This problem is parametric in the arities and polarities of the non-constant, the partial order on constants $(B, B_{\leq})$, and whether or not $\{\bot, \top\}$ is included in the signature.

We present the proof in four steps. We first show how to reduce structural subtype satisfiability to 1-subtype satisfiability (Section 4.3.1) and then do the same for the non-structural case (Section 4.3.2). Next, we reduce 1-subtype satisfiability to uniform subtype satisfiability (Section 4.3.3). Finally, we translate uniform subtype satisfiability back to both structural and non-structural subtype satisfiability (Section 4.3.4).

4.3.1 Structural to 1-subtype satisfiability

In this part, we show how to reduce structural to 1-subtype satisfiability. We first use a standard technique to characterize the shapes of solutions to a structural subtype constraints. We consider type expressions as terms over Σ.

Definition 4.1
Let φ be a constraint over Σ and $\star$ be an arbitrary, fixed constant. For any type expression t, $t^\star$ denotes the same type expression as t except all constants in t are replaced with $\star$. Define the shape constraint *$sh(\varphi)$ as:*

$$\begin{aligned} sh(t_1 = t_2) &=_{\text{df}} t_1^\star = t_2^\star \\ sh(x \leq y) &=_{\text{df}} x = y \\ sh(\varphi_1 \wedge \varphi_2) &=_{\text{df}} sh(\varphi_1) \wedge sh(\varphi_2) \end{aligned}$$

The constraint φ is called weakly unifiable *iff $sh(\varphi)$ is unifiable.*

Consider a signature $\Sigma = B \cup \{\times, \rightarrow\}$. We construct a signature $s(\Sigma) =_{\text{df}} B \cup \{f, c\}$, where f is function symbol of arity four and c is a fresh constant. Our approach is to use f to capture both $\times$ and $\rightarrow$, *i.e.*, all the non-constant function symbols in Σ. The first two arguments of f are used to model the two arguments of $\times$ and the next two to model the two arguments of $\rightarrow$. Thus, f is co-variant in all arguments except the third one.

Given a constraint φ over Σ, we construct $s(\varphi)$ over $s(\Sigma)$:

$$\begin{array}{rcll} s(x = y \times z) & =_{\text{df}} & x = f(y, z, c, c) & \\ s(x = y \rightarrow z) & =_{\text{df}} & x = f(c, c, y, z) & \\ s(x = b) & =_{\text{df}} & x = b & \forall b \in B \\ s(x \leq y) & =_{\text{df}} & x \leq y & \\ s(\varphi_1 \wedge \varphi_2) & =_{\text{df}} & s(\varphi_1) \wedge s(\varphi_2) & \end{array}$$

Lemma 4.11
If φ is weakly unifiable, then φ is satisfiable over Σ iff $s(\varphi)$ is satisfiable over $s(\Sigma)$.

Its proof follows from Lemma 3.9 given in Section 3.3.

4.3.2 Non-structural to 1-subtype satisfiability

We handle non-structural signatures $\Sigma = B \cup \{\bot, \top, \times, \rightarrow\}$, similarly. The new signature is defined in exactly the same way as for the structural case by $s(\Sigma) = B \cup \{\bot, \top, f, c\}$. Constraints are also transformed in the same way, except including two extra rules for $\bot$ and $\top$:

$$\begin{array}{rcl} s(x = \bot) & =_{\text{df}} & x = \bot \\ s(x = \top) & =_{\text{df}} & x = \top \end{array}$$

However, weak unifiability is not sufficient for the initial satisfiability check. To see that, consider, for example, $x \leq y \times z \wedge x \leq u \rightarrow v$, which is satisfiable, but not weakly unifiable. To address this problem, we introduce a notion of *weak satisfiability*. It is similar to weak unifiability, except subtype ordering is also retained.

Definition 4.2
Let φ be a constraint over Σ, and $\star$ be an arbitrary and fixed constant. We define $t^\star$ as before, except $\bot^\star =_{\text{df}} \bot$ and $\top^\star =_{\text{df}} \top$. Define the weak satisfiability constraint *$ws(\varphi)$ as:*

$$\begin{aligned} ws(t_1 = t_2) &=_{\text{df}} t_1^\star = t_2^\star \\ ws(x{\leq}y) &=_{\text{df}} x{\leq}y \\ ws(\varphi_1 \wedge \varphi_2) &=_{\text{df}} ws(\varphi_1) \wedge ws(\varphi_2) \end{aligned}$$

The constraint φ is called weakly satisfiable *iff $ws(\varphi)$ is satisfiable.*

Lemma 4.12
If φ is weakly satisfiable, then φ is satisfiable over Σ iff $s(\varphi)$ is satisfiable over $s(\Sigma)$.

The proof of this lemma requires the following result. Let φ be a constraint over a non-structural signature Σ. If $ws(\varphi)$ is satisfiable, then $ws(\varphi)$ has a minimum shape solution α by a simple extension of a theorem of Palsberg, Wand and OKeefe on non-structural subtype satisfiability over lattices [PWO97]. We claim that if φ is satisfiable, then φ also has a minimum shape solution that is of the same shape as α.

Lemma 4.13
If φ is satisfiable over Σ, let α be a minimum shape solution for $ws(\varphi)$, and in addition, α is such a solution with the least number of leaves assigned $\star$. Then φ has a solution β that is of the same shape as α, i.e., *for all $x \in V(\varphi) = V(ws(\varphi))$, $sh(\alpha(x) = \beta(x))$ is unifiable. Furthermore, β is a minimum shape solution of φ.*

PROOF. Let γ be a solution for φ. We construct a variable assignment β for φ from α and γ:

$$\beta(x)(\pi) =_{\text{df}} \begin{cases} \gamma(x)(\pi) & \text{if } \alpha(x)(\varphi) = \star \\ \alpha(x)(\pi) & \text{otherwise.} \end{cases}$$

One can show that $dom(\alpha) = dom(\beta)$, because if $\alpha(x)(\pi) = \star$, $\gamma(x)(\pi)$ must be a constant. We verify that $\beta \models \varphi$:

- Consider a literal of the form $x = f(x_1, \ldots, x_n)$ for $n > 0$. Both $\alpha(x) = f(\alpha(x_1), \ldots, \alpha(x_n))$ and $\gamma(x) = f(\gamma(x_1), \ldots, \gamma(x_n))$. Thus, if $\alpha(x)(\pi) = \star = (f(\alpha(x_1), \ldots, \alpha(x_n))(\pi)$, $\beta(x)(\pi) = \gamma(x)(\pi) = f(\gamma(x_1), \ldots, \gamma(x_n))(\pi) = f(\beta(x_1), \ldots, \beta(x_n))(\pi)$. Otherwise, $\beta(x)(\pi) = \alpha(x)(\pi) = f(\alpha(x_1), \ldots, \alpha(x_n))(\pi) = f(\beta(x_1), \ldots, \beta(x_n))(\pi)$.
- Consider a literal of the form $x = b$ for $b \in B \cup \{\bot, \top\}$. Clearly, $\beta(x) = b$.
- Consider a literal of the form $x{\leq}y$. We use a simple case analysis on the possible values of $\alpha(x)(\pi)$ and $\alpha(y)(\pi)$ for each address π.

Lemma 3.9 and Lemma 4.13 together imply the following corollary, which is used next in Section 4.4 to treat subtype satisfiability interpreted over finite trees.

Corollary 4.1
A subtype constraint φ is satisfiable over finite trees if and only if φ is satisfiable over finite trees of height bounded by $|\varphi|$. This holds for both structural and non-structural signatures.

4.3.3 1-subtype to uniform satisfiability

In this part, we give a reduction from 1-subtype to uniform subtype satisfiability. This reduction is uniform for subtyping with and without $\bot$ and $\top$.

Proposition 4.6
Over possibly infinite trees, 1-subtype satisfiability is linear time reducible to uniform subtype satisfiability.

PROOF. Let Σ be a 1-subtype signature. We define a uniform signature $s(\Sigma)$ by extending the arities of all function symbols to the maximal arity of Σ (*i.e.*, the arity of the only non-trivial function symbol), such that:

- $s(\Sigma) =_{\mathrm{df}} \Sigma$;
- $\forall f \in s(\Sigma).\mathit{arity}_{s(\Sigma)}(f) =_{\mathrm{df}} \mathrm{max}$;
- $\leq_{s(\Sigma)} =_{\mathrm{df}} \leq_{\Sigma}$,

where max is the maximal arity of Σ.

We next translate a subtype constraint φ over Σ to a constraint $s(\varphi)$ over $s(\Sigma)$:

$$s(x{=}f(x_1,\ldots,x_{\max})) =_{\mathrm{df}} x{=}f(x_1,\ldots,x_{\max}) \quad (4.1)$$
$$s(x{=}b) =_{\mathrm{df}} x{=}b(y_1,\ldots,y_{\max}) \quad (4.2)$$
$$s(x_1 \leq x_2) =_{\mathrm{df}} x_1 \leq x_2 \quad (4.3)$$
$$s(\varphi_1 \wedge \varphi_2) =_{\mathrm{df}} s(\varphi_1) \wedge s(\varphi_2) \quad (4.4)$$
$$s(x{=}\bot) =_{\mathrm{df}} x{=}\bot(u_1,\ldots,u_{\max}) \quad (4.5)$$
$$s(x{=}\top) =_{\mathrm{df}} x{=}\top(v_1,\ldots,v_{\max}) \quad (4.6)$$

where the y_i's, u_i's, and v_i's are fresh variables, and rules (5) and (6) are additional ones for a non-structural signature.

Lemma 4.14
A subtype constraint φ over a standard signature Σ is satisfiable if and only if $s(\varphi)$ is satisfiable over the uniform signature $s(\Sigma)$.

PROOF of ($\Leftarrow$). For this implication, we define a transformation of $cut : tree_{s(\Sigma)} \to tree_\Sigma$:

- $cut(f(\tau_1, \ldots, \tau_{\max})) =_{\text{df}} f$, if $f \in \Sigma_0$;
- $cut(f(\tau_1, \ldots, \tau_{\max})) =_{\text{df}} f(cut(\tau_1), \ldots, cut(\tau_{\max}))$, otherwise.

We fix a solution α of $s(\varphi)$ and show that the variable assignment $cut \circ \alpha$ satisfies φ over Σ. We need to verify that $cut \circ \alpha$ satisfies all literals of φ:

1. Consider a literal of the form $x{=}f(x_1, \ldots, x_{\max})$ in φ (for $arity(f) = \max$). We know that $\alpha \models x = f(x_1, \ldots, x_{\max})$. In addition, the following sequence of implications holds:

$$\begin{array}{ll} & \alpha \models x{=}f(x_1, \ldots, x_{\max}) \\ \Leftrightarrow & \alpha(x) = f(\alpha(x_1), \ldots, \alpha(x_{\max})) \\ \Rightarrow & cut(\alpha(x)) = f(cut(\alpha(x_1)), \ldots, cut(\alpha(x_{\max}))) \\ \Leftrightarrow & cut \circ \alpha \models x{=}f(x_1, \ldots, x_{\max}) \end{array}$$

2. Consider a literal of the form $x{=}b$ in φ. We know $\alpha \models x = b(y_1, \ldots, y_{\max})$ for some fresh variables y_i. Similarly, we have the following sequence of implications:

$$\begin{array}{ll} & \alpha \models x{=}b(y_1, \ldots, y_{\max}) \\ \Leftrightarrow & \alpha(x) = b(\alpha(y_1), \ldots, \alpha(y_{\max})) \\ \Rightarrow & cut(\alpha(x)) = b \\ \Leftrightarrow & cut \circ \alpha \models x{=}b \end{array}$$

3. Consider a literal of the form $x \leq y$ in φ. We know $D_2 = dom(cut(\alpha(x))) \cap dom(cut(\alpha(y))) \subseteq dom(\alpha(x)) \cap dom(\alpha(y)) = D_1$. We thus have:

$$\begin{array}{ll} & \alpha \models x \ \leq y \\ \Leftrightarrow & \alpha(x) \ \leq \alpha(y) \\ \Leftrightarrow & \forall \pi \in D_1.\alpha(x)(\pi) \ \leq_\Sigma \alpha(y)(\pi) \\ \Rightarrow & \forall \pi \in D_2.cut(\alpha(x)(\pi)) \ \leq_\Sigma cut(\alpha(y)(\pi)) \\ \Leftrightarrow & cut \circ \alpha \models x \ \leq y \end{array}$$

4. For a non-structural signature, literals of the form $x{=}\bot$ and $x{=}\top$ are treated similarly as $x{=}b$ in the second case.

PROOF of ($\Rightarrow$) We now consider the inverse implication. We first define a mapping $ext : tree_\Sigma \to tree_{s(\Sigma)}$:

- $ext(f(\tau_1, \ldots, \tau_{\max})) =_{\text{df}} f(ext(\tau_1), \ldots, ext(\tau_{\max}))$
- $ext(b) =_{\text{df}} b(\star^\infty, \ldots, \star^\infty)$, where $\star$ is an arbitrary, fixed constant in B of Σ, and $\star^\infty$ denotes the complete, infinite tree where each node is labeled $\star$, *i.e.*, the unique solution to the equation $x = \star(x, \ldots, x)$.
- For a non-structural signature, $ext(\bot)$ and $ext(\top)$ are defined respectively as the smallest tree and the greatest tree (over the two max-ary symbols $\bot$ and $\top$ in $s(\Sigma)$). They are defined mutually recursively and are unique solutions to the following equations:

  $$\begin{aligned} x &= \bot(x_1, \ldots, x_{\max}) \\ y &= \top(y_1, \ldots, y_{\max}) \end{aligned}$$

  where $x_i = x$ and $y_i = y$ if the i-th argument is co-variant; and $x_i = y$ and $y_i = x$ otherwise.

  As an example, for a standard signature with the function type constructor $\to$, $ext(\bot)$ and $ext(\top)$ give the unique solution to the equations $x = \bot(y, x)$ and $y = \top(x, y)$.

We first state and prove the following lemma regarding ext.

Lemma 4.15
If $\tau_1 \leq \tau_2$, then $ext(\tau_1) \leq ext(\tau_2)$.

PROOF. We use a proof by contradiction. We prove for a non-structural signature. For structural signatures, the proof is exactly the same, except discarding all cases involving $\bot$ or $\top$.

For $\tau_1 \leq \tau_2$, assume that $ext(\tau_1) \not\leq ext(\tau_2)$. Then there is a *shortest* path π such that $ext(\tau_1)(\pi) \not\leq^\pi ext(\tau_2)(\pi)$. Clearly, $\pi \neq \varepsilon$, and we let $\pi = \pi'.i$ for some π' and $i \in \{1, \ldots, \max\}$. We have a few cases:

1. When $\pi \in dom(\tau_1) \wedge \pi \in dom(\tau_2)$: This case is impossible because it contradicts the assumption that $\tau_1 \leq \tau_2$.
2. When $\pi \in dom(\tau_1) \wedge \pi \notin dom(\tau_2)$: $ext(\tau_2)(\pi')$ must be $\top$ if $par(\pi') = 1$ or $\bot$ if $par(\pi') = -1$. In either case, it is clear that $ext(\tau_1)(\pi) \leq^\pi ext(\tau_2)(\pi)$, a contradiction.
3. When $\pi \notin dom(\tau_1) \wedge \pi \in dom(\tau_2)$: This case is symmetric to the previous one.

4. When $\pi \notin dom(\tau_1) \wedge \pi \notin dom(\tau_2)$: We know $ext(\tau_1)(\pi')$ and $ext(\tau_2)(\pi')$ must be constants. In addition, $ext(\tau_1)(\pi') \leq^{\pi'} ext(\tau_2)(\pi')$, because π is the shortest path such that $ext(\tau_1)(\pi) \not\leq^{\pi} ext(\tau_2)(\pi)$. This would, however, imply that $ext(\tau_1)(\pi) \leq^{\pi} ext(\tau_2)(\pi)$, a contradiction.

We can now finish the proof of Lemma 4.14. Let $\alpha \models \varphi$. We construct a variable assignment β for $s(\varphi)$:

- $\beta(x) =_{\text{df}} ext(\alpha(x))$ for all variables $x \in V(\varphi)$;
- $\beta(x) =_{\text{df}} \star^{\infty}$ for the y_i's;
- $\beta(x) =_{\text{df}} ext(\bot)$ for a co-variant u_i or contra-variant v_i;
- $\beta(x) =_{\text{df}} ext(\top)$ for a contra-variant u_i or co-variant v_i.

We need to show that $\beta \models s(\varphi)$. There are a few kinds of literals in $s(\varphi)$:

1. Consider a literal of the form $x = f(x_1, \ldots, x_{\max})$ of $s(\varphi)$, derived from $x = f(x_1, \ldots, x_{\max})$ of φ. We know that $\alpha \models x = f(x_1, \ldots, x_{\max})$, *i.e.*, $\alpha(x) = f(\alpha(x_1), \ldots, \alpha(x_{\max}))$. Thus, we have:

$$\begin{aligned} \beta(x) &= ext(f(\alpha(x_1), \ldots, \alpha(x_{\max}))) \\ &= f(ext(\alpha(x_1)), \ldots, ext(\alpha(x_{\max}))) \\ &= f(\beta(x_1), \ldots, \beta(x_{\max})) \\ &= \beta(f(x_1, \ldots, x_{\max})) \end{aligned}$$

   Hence, $\beta \models x = f(x_1, \ldots, x_{\max})$.

2. Consider a literal of the form $x = b(y_1, \ldots, y_{\max})$, derived from $x = b$ in φ. We know that $\alpha \models x = b$, *i.e.*, $\alpha(x) = b$. We thus have:

$$\begin{aligned} \beta(x) &= ext(\alpha(x)) \\ &= ext(b) \\ &= b(\beta(y_1), \ldots, \beta(y_{\max})) \\ &= \beta(b(y_1, \ldots, y_{\max})) \end{aligned}$$

   Hence, $\beta \models x = b(y_1, \ldots, y_{\max})$.

3. Consider a literal of the form $x \leq y$, derived from $x \leq y$ in φ. We know $\alpha \models x \leq y$, *i.e.*, $\alpha(x) \leq \alpha(y)$. By Lemma 4.15, we have $\beta(x) = ext(\alpha(x)) \leq ext(\alpha(y)) = \beta(y)$. Thus, $\beta \models x \leq y$.

4. For a non-structural signature, we have literals of the form $x{=}\bot(u_1, \ldots, u_{\max})$ and $x{=}\top(v_1, \ldots, v_{\max})$. For the case with $\bot$, we know $\alpha \models x = \bot$. We thus have:

$$\begin{aligned} \beta(x) &= ext(\alpha(x)) \\ &= ext(\bot) \\ &= \bot(\beta(u_1), \ldots, \beta(u_{\max})) \\ &= \beta(\bot(u_1, \ldots, u_{\max})) \end{aligned}$$

   The case for $\top$ is similar.

With Lemma 4.14, we have finished the proof of Proposition 4.6.

4.3.4 Uniform to (non-)structural satisfiability

In this part, we prove the last step of the equivalence (Theorem 4.4), namely, how to reduce uniform subtype satisfiability to structural and non-structural subtype satisfiabilities.

Proposition 4.7
Uniform subtype satisfiability is linear time reducible to structural and non-structural subtype satisfiability over possibly infinite trees.

To simplify its proof we assume a uniform subtype problem where all function symbols have arity three with their first two arguments being contravariant and the last one covariant. This proof can be easily adapted to uniform signatures with other arities and polarities.

We construct a reverse translation $\bar{\bar{s}}$ of s (defined in Section 4.3.3) in two steps. Let Σ be a uniform signature with symbols of arity three. We first define a standard signature $\bar{s}(\Sigma)$ by including symbols in Σ as constants and adding $\to$:

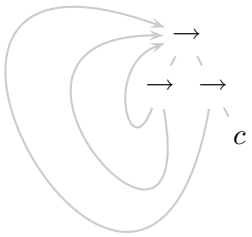
$$\begin{aligned} \bar{s}(\Sigma) &=_{\text{df}} \Sigma \cup \{\to\} \\ \forall g \in \Sigma.\ arity_{\bar{s}(\Sigma)}(g) &=_{\text{df}} 0 \\ arity_{\bar{s}(\Sigma)}(\to) &=_{\text{df}} 2 \\ \leq_{\bar{s}(\Sigma)} &=_{\text{df}} \leq_{\Sigma} \end{aligned}$$

We now translate a subtype constraint φ over Σ to a constraint $\bar{s}(\varphi)$ over $\bar{s}(\Sigma)$:

$$\begin{array}{rcl} \bar{s}(x{=}g(x_1,x_2,x_3)) & =_{\text{df}} & x{=}(x_3 \to x_2) \to (x_1 \to g) \\ \bar{s}(x_1 \leq x_2) & =_{\text{df}} & x_1 \leq x_2 \\ \bar{s}(\varphi_1 \wedge \varphi_2) & =_{\text{df}} & \bar{s}(\varphi_1) \wedge \bar{s}(\varphi_2) \end{array}$$

where we use a non-flat constraint in the first line for a simpler presentation. The arguments x_1, x_2 are again contravariant and x_3 is covariant in the constraint $\bar{s}(x{=}g(x_1,x_2,x_3))$. Thus, $\bar{s}$ preserves all polarities.

In our second step, we force every variable to be mapped to the following fixed, infinite shape:



Therefor, we extend $\bar{s}(\Sigma)$ to $\bar{\bar{s}}(\Sigma)$ with four new constants a_1, a_2, a_3, and a_4 with the following ordering: $a_1 \leq c \leq a_3 \ \wedge \ a_2 \leq c \leq a_4$, for all constants $c \in \bar{s}(\Sigma)$. We define $\bar{\bar{s}}(\varphi)$ as the conjunction of $\bar{s}(\Sigma)$ and the following constraints:

(1) $u_1 \leq x \wedge u_2 \leq x \wedge x \leq u_3 \wedge x \leq u_4$, for each variable $x \in V(\bar{s}(\Sigma))$;

(2) $\bigwedge_{i=1,2,3,4} \ u_i{=}(u_i \to u_i) \to (u_i \to a_i)$

The constraints (1) and (2) in $\bar{\bar{s}}(\varphi)$ determine the shape of any variable $x \in V(\bar{s}(\varphi))$. We claim, in the following lemma, that any solution to $\bar{\bar{s}}(\varphi)$ must be of a particular shape and must also map variables $x \in V(\bar{s}(\varphi))$ to trees over $\bar{s}(\Sigma)$.

Lemma 4.16
When the constraint $\bar{\bar{s}}(\varphi)$ is interpreted over any (non-)structural signature $\bar{\bar{s}}(\Sigma)$ or $\bar{\bar{s}}(\Sigma) \cup \{\bot, \top\}$, every variable assignment $\alpha \models \bar{\bar{s}}(\varphi)$ satisfies that for all paths $\pi \in (1(1 \cup 2) \cup 21)^*$:

$$\begin{array}{rcll} \alpha(x)(\pi') & = & \to & \text{if } \pi' \text{ is a prefix of } \pi \\ \alpha(x)(\pi 22) & = & \left\{ \begin{array}{ll} a_i & \text{if } x = u_i \\ c \in \Sigma & \text{otherwise.} \end{array} \right. & \end{array}$$

PROOF by definition of $\bar{\bar{s}}(\varphi)$.

Lemma 4.17
A subtype constraint φ over a uniform signature Σ is satisfiable if and only if the constraint $\bar{\bar{s}}(\varphi)$ over $\bar{\bar{s}}(\Sigma)$ is satisfiable. This statement also holds if we replace the structural signature $\bar{\bar{s}}(\Sigma)$ by the non-structural signature $\bar{\bar{s}}(\Sigma) \cup \{\bot, \top\}$.

PROOF. We define a transformation of $map : tree_\Sigma \to tree_{\bar{\bar{s}}(\Sigma)}$ on trees for all $g \in \Sigma$:

$$\begin{aligned} map(g(\tau_1, \tau_2, \tau_3)) \ =_{\text{df}} \quad & (map(\tau_3) \to map(\tau_2)) \\ \to \quad & (map(\tau_1) \to g) \end{aligned}$$

With that it can be easily verified that if there exists a solution $\alpha \models \varphi$ over an uniform signature Σ then $map(\alpha) \models \bar{\bar{s}}(\varphi)$ holds over $\bar{\bar{s}}(\Sigma)$. For the other direction we assume an assignment $\alpha \models \bar{\bar{s}}(\varphi)$. Then there also exists an assignment $\beta = map^{-1}(\alpha)$ according to the shape of any solution of $\bar{\bar{s}}(\varphi)$ stated in Lemma 4.16. Again, it can be easily verified that $\beta \models \Sigma$.

The proof also holds in the case where we add $\bot$ and $\top$ to $\bar{\bar{s}}(\Sigma)$ since both symbols cannot occur in any node of any solution of $\bar{\bar{s}}(\Sigma)$ (again Lemma 4.16).

4.4 Finite subtype satisfiability over posets

The complexity of finite structural subtype satisfiability was shown to be *PSPACE*-complete by Tiuryn [Tiu92] and Frey [Fre02]. Here, we establish the same complexity for the non-structural case.

4.4.1 PSPACE-hardness

Proposition 4.8
Non-structural subtype satisfiability over finite trees is PSPACE-hard.

The analogous result for the structural case was shown by Tiuryn [Tiu92]). To lift this result, we show how to reduce non-structural to structural subtype satisfiability.

Lemma 4.18
Structural subtype satisfiability is polynomial time reducible to non-structural subtype satisfiability (both for finite and infinite trees).

PROOF. Let Σ be a structural signature. We construct a non-structural signature:

$$s(\Sigma) \ =_{\text{df}} \ \Sigma \cup \{\bot, \top, a_1, a_2, a_3, a_4\}$$

with the a_i's four new constants. In addition,

$$\leq_{s(\Sigma)} =_{\mathrm{df}} \leq_\Sigma \cup\{(a_1,c),(a_2,c),(c,a_3),(c,a_4) \mid c \in \Sigma_0\}$$

Let φ be a constraint over Σ. We construct $s(\varphi)$ over $s(\Sigma)$. Consider φ's shape constraint $sh(\varphi)$ (see Definition 4.1). If $sh(\varphi)$ is not unifiable, we simply let $s(\varphi) =_{\mathrm{df}} \top\leq\bot$. Otherwise, consider the most general unifier (m.g.u.) γ of $sh(\varphi)$. We let $sh(\varphi)'$ be the same as $sh(\varphi)$ except each occurrence of $\star$ is replaced with a fresh variable. We make two copies of $sh(\varphi)'$, $sh(\varphi)'_L$ and $sh(\varphi)'_R$ (for left and right), where each variable x is distinguished as x_L and x_R respectively. For each variable $x \in V(\varphi)$, if $\gamma(x)$ is either $\star$ or belongs to $V(\varphi)$, we say x is atomic. For a variable x, let $force(x)$ denote the constraint:

$$a_1\leq x \wedge a_2\leq x \wedge x\leq a_3 \wedge x\leq a_4.$$

Notice that Lemma 4.18 holds both for finite and infinite trees.

We can now construct $s(\varphi)$, which is the conjunction of the following components: (1) φ itself; (2) $sh(\varphi)'_L$; (3) $sh(\varphi)'_R$; (4) For each atomic $x \in V(\varphi)$, $force(x_L)$ and $force(x_R)$; (5) For each fresh variable x in $sh(\varphi)'_L$ and $sh(\varphi)'_R$, $force(x)$; and (6) For each variable $x \in V(\varphi)$, $x_L\leq x\leq x_R$. One can show that φ is satisfiable over Σ iff $s(\varphi)$ is satisfiable over $s(\Sigma)$.

4.4.2 A PSPACE algorithm

THEOREM 4.5
Finite non-structural subtype satisfiability is PSPACE-complete.

It remains to prove membership in *PSPACE* which subsumes the proof of Frey [Fre02]. We present a new proof idea based on *K-normal modal logic* which applies uniformly to the non-structural and the structural cases.

We assume a signature Σ that contains at least one constant c and non-constant symbol f. Satisfiability would be trivial otherwise. We adapt our reduction to uniform signature for the finite case. Let $s(\Sigma)$ be the uniform signature for Σ. With the following formulas we define an additional subtype ordering $\sqsubset_{s(\Sigma)}$ by $g\sqsubset_{s(\Sigma)}f$ for all $g \in s(\Sigma)$ and one fixed symbol f:

$$\begin{array}{lll} \mathit{finite}_n(x) & =_{\mathrm{df}} & \exists y_1 ... \exists y_{n+1}.\ x\sqsubseteq y_1 \\ & \wedge & \wedge_{i=1}^{n} y_i = f(y_{i+1}, ... y_{i+1}) \\ & \wedge & y_{n+1} = c(y_{n+1}, ... y_{n+1}) \\ \mathit{finite}_n(V) & =_{\mathrm{df}} & \wedge_{x\in V}\, \mathit{finite}_n(x) \end{array}$$

The following proposition holds by Theorem 4.4 and Corollary 4.1.

Proposition 4.9
A subtype constraint φ is satisfiable over finite Σ-trees if and only if the constraint $s(\varphi) \wedge \mathit{finite}_{|\varphi|}(V(\varphi))$ is satisfiable over uniform signatures $s(\Sigma)$.

PROOF by Theorem 4.4 and Corollary 4.1.

We can easily adapt the translation of subtype constraints over uniform signatures into inverted core PDL_n from Table 4.4 to handle the formulas $\mathit{finite}_{|\varphi|}(V(\varphi))$ as well. This yields a satisfiability preserving encoding into inverted core PDL_n for the finite case. We finally alter this encoding to a translation into the following modal logic:

$$D \quad ::= \quad B \mid [\{1,\ldots,n\}^{|\varphi|}]B \mid D_1 \wedge D_2$$

Because all trees in finite solutions of φ have at most linear depth it is correct to replace all $[*]$ modalities by $[\{1,\ldots,n\}^{|\varphi|}]$, both, in the reduction of Table 4.4 and in the well-formedness property wff_V. This gives a translation into formulas D.

Proposition 4.10
Satisfiability of inverted linearly depth-bounded PDL_n formulas D is in PSPACE.

PROOF. We translate the problem to satisfiability of *K-normal modal logic* over the complete infinite n-ary tree (which is known to be in *PSPACE* [Spa93]). It is defined by the syntax of PDL_n formulas A restricted to the single modality $[\{1,2\}]$:

$$E \quad ::= \quad p \mid \neg E \mid E \wedge E' \mid E \leftrightarrow E' \mid \Box E$$

We denote i repetitions of $\Box$ by $\Box^i$. The only complication is to translate formulas $[i]B$. In the case of binary trees (other cases of n are analogous) we do so by translating $[1]B$ to $\Box(p \rightarrow B)$ and $[2]B$ to $\Box(\neg p \rightarrow B)$, where we use a new variable p that is true at all paths $\pi 1$ and false at paths $\pi 2$. Following [Hem00] p can be axiomatized by:

$$\wedge_{i=1}^{m} \Box^{i-1}(\langle\{1,2\}\rangle\Box^{m-i}p \wedge \langle\{1,2\}\rangle\Box^{m-i}\neg p).$$

4.5 Summary

We have given a complete characterization of the complexity of subtype satisfiability over posets through a new connection of subtype satisfiability with modal logics, which have well understood satisfiability problems. Our technique yields a uniform and

systematic treatment of different choices of subtype orderings: finite versus recursive types, structural versus non-structural subtyping, and considerations of symbols with co- and contra-variant arguments.

Part II

Non-Structural Subtype Entailment

Non-Structural Entailment (NSSE)

We investigate non-structural subtype entailment (NSSE) for a restricted signature $\{\bot, \top, f\}$. Our contribution to entailment yields a new characterization of NSSE that uses regular expressions and word equations [Mak77, Pla99]. More precisely we map NSSE to the question whether so called cap set expressions do contain all words over an alphabet (universality problem). In a next step we translate the universality of certain cap expressions also back to NSSE. For the proof of both directions we introduce new automata: cap automata which exactly recognize the same languages as cap set expressions. Our NSSE characterization also based on the satisfiability tests for subtype constraints which we have given in Chapter 3. Finally, we prove a fragment of NSSE to be decidable by reducing it to satisfiability of word equations with regular constraints.

5 Characterization of NSSE

We now formulate our main result and discuss its relevance (Theorem 5.1). This is a new characterization of NSSE which is based on a new class of extended regular expressions: cap set expressions *that we introduce first.*

We start with *regular expressions* R over some alphabet A that are defined as usual:

$$R := a \mid \varepsilon \mid R_1 R_2 \mid R^* \mid R_1 \cup R_2 \mid \emptyset \qquad \text{where } a \in A$$

Every regular expression R describes a regular language of words $\mathcal{L}(R) \subseteq A^*$. We next introduce *cap set expressions* E over A. (Their name will be explained in Section 6.2.)

$$E \quad ::= \quad R_1 R_2^\circ \mid \; E_1 \cup E_2$$

Cap set expressions E denote sets of words $\mathcal{L}(E) \subseteq A^*$ that we call *cap sets*. We have to define the *cap set operator* $^\circ$ on sets of words, i.e., we must define the set $S^\circ \subseteq A^*$ for all sets $S \subseteq A^*$. Let pr be the prefix operator lifted to sets of words. We set:

$$S^\circ \;=\; \{\pi \mid \pi \in pr(\mu^*),\ \mu \in S\}$$

A word π belongs to S° if π is a prefix of a power $\mu \ldots \mu$ of some word $\mu \in S$. Note that cap set expressions subsume regular expressions: indeed, $\mathcal{L}(R) = \mathcal{L}(R\,\varepsilon^\circ)$ for all R. But the cap operator adds new expressiveness when applied to an infinite set: there exist regular expression R such that the language of the cap set expression R° is neither regular nor context free. Consider for instance $(21^*)^\circ$ which denotes the set of all prefixes of words $21^n\,21^n \ldots 21^n$ where $n \geq 0$. Clearly this set is not context-free.

We will derive appropriate *restrictions* on cap set expressions (Def. 8.1) such that the following theorem becomes true.

THEOREM 5.1 (Characterization)
The decidability of NSSE for a signature $\{\bot, f, \top\}$ with a single function symbol of arity $n \geq 1$ is equivalent to the decidability of the universality problem for the class of restricted cap set expressions over the alphabet $\{1, \ldots, n\}$. This result holds equally for finite, regular, and possibly infinite trees.

The theorem allows us to derive the following robustness result of NSSE against variations from automata transformations (Section 10.1).

Corollary 5.1
All variants of NSSE with signature $\{\bot, f, \top\}$ where the arity of f is at least $n \geq 2$ are equivalent. It does not even matter whether finite, regular, or infinite trees are considered.

Theorem 5.1 can also be used to relate NSSE to word equations. The idea is to express membership in cap sets in the positive existential fragment of word equations with regular constraints [Sch91]. The reduction can easily be based on the following lemma that is well known in the field of string unification.

Lemma 5.1
For all words $\pi \in A^*$ and nonempty words $\mu \in A^+$ it holds that $\pi \in pr(\mu^*)$ if and only if $\pi \in pr(\mu\pi)$.

PROOF. If $\pi \in pr(\mu^*)$ then there is a natural number $n \geq 1$ such that $\mu^{n-1} \leq \pi \leq \mu^n$. Hence, $\pi \leq \mu\mu^{n-1} \leq \mu\pi$ as required. For the converse, let $\pi \leq \mu\pi$ where $\mu \neq \varepsilon$. We prove $\pi \in pr(\mu^*)$ by induction on the length of π. If $|\pi| \leq |\mu|$ then $\pi \leq \mu$ and thus $\pi \in pr(\mu^*)$ as required. Otherwise, there exists a path π' with $\pi = \mu\pi'$ and thus $\pi' \leq \mu\pi'$ by our assumption that $\pi \leq \mu\pi$. Note that π' is a proper prefix of π since $\mu \neq \varepsilon$. The induction hypothesis applied to π' yields $\pi' \in pr(\mu^*)$ such that $\pi \in pr(\mu^*)$.

Lemma 5.1 states that all sets $S \subseteq A^+$ of nonempty words satisfy:

$$\pi \in S^\circ \leftrightarrow \exists\mu\exists\nu\ (\mu \in S \wedge \pi\nu{=}\mu\pi)$$

Theorem 5.1 thus implies that we can express the universality problem of cap set expressions E in the positive $\forall\exists^*$ fragment of the first-order theory of word equations with regular constraints.

Corollary 5.2
NSSE with a single function symbol of arity $n \geq 1$ can be expressed in the positive $\forall\exists^*$ fragment of the first-order theory of word equations with regular constraints over the alphabet $\{1, \ldots, n\}$.

Unfortunately, even the positive $\forall\exists^3$ fragment of a single word equation is undecidable [Dur95] except if the alphabet is infinite [BS96] or a singleton [VR83]. Therefore, it remains open whether NSSE is decidable or not. But it becomes clear that the difficulty is raised by word equations hidden behind cap set expressions R°, i.e. the equation $\pi\nu{=}\mu\pi$ in Lemma 5.1.

Theorem 5.1 constitutes a promising starting point to further investigate decidability of NSSE. For instance, we can infer a new decidability result for the monadic case directly from Corollary 5.2.

Corollary 5.3
NSSE is decidable for the signature $\{\bot, f, \top\}$ *if* f *is unary.*

6 Cap Automata and Cap Sets

6.1 Safety

The goal of this section is to characterize NSSE by properties of sets of words, that we call safety properties. Appropriate safety properties can be verified by P-automata as we will show in Chapter 7.

We use terms $x(\pi)$ to denote the node label of the value of x at path π. Whenever we use this term, we presuppose the existence of π in the tree domain of the value of x. For instance, the formula $x(12) \leq_\Sigma \top$ is satisfied by a variable assignment if and only if the tree assigned to x contains the node 12.

We next recall the notion of safety from [NP99]. Let $\varphi \models^? x{\leq}y$ be an entailment judgment and π a word in $\{1, \ldots, \mathit{arity}(f)\}^*$. We call π *safe* for $\varphi \models^? x{\leq}y$ if entailment cannot be contradicted at π, i.e. if $\varphi \wedge y(\pi) <_\Sigma x(\pi)$ is unsatisfiable. Clearly entailment $\varphi \models x{\leq}y$ is equivalent to that all paths are safe for $\varphi \models^? x{\leq}y$.

For a restricted class of entailment judgments it is shown in [NP99] that the above notion of safety can be checked by testing universality of P-automata. Unfortunately, it is unclear how to lift this result to the general case. To work around, we will refine the notion of safety into two dual notions: *left (l) safety* and *right (r) safety*. These notions will be defined on formulas:

$$\mathrm{pref}^g_\pi(x) \quad =_{\mathrm{def}} \quad \bigvee_{\pi' \leq \pi} x(\pi') =_\Sigma g$$

for some function symbol $g \in \Sigma$, word π, and variable x. It requires x to denote a tree that is labeled by g at some prefix π' of π. We now define l-safety for words $\pi \in \{1, \ldots, \mathit{arity}(f)\}^*$ with respect to judgments $\varphi \models^? x{\leq}y$:

$$\pi \text{ is } \textit{l-safe} \text{ for } \varphi \models^? x{\leq}y \quad \text{iff} \quad \varphi \models \mathrm{pref}^\top_\pi(x) \rightarrow \mathrm{pref}^\top_\pi(y)$$

If π is l-safe for judgment $\varphi \models^? x{\leq}y$ then entailment contradicted by a solution α of φ that maps the left hand side x to some tree where $\alpha(x)(\pi) =_\Sigma \top$, except if

$\alpha(x)(\pi') =_\Sigma \top$ and $\alpha(y)(\pi'') =_\Sigma \top$ for some prefixes π' and π'' of π. The notion of r-safety is analogous; here one tries to contradict with $\bot$ at the right hand side y:

$$\pi \text{ is } \textit{r-safe} \text{ for } \varphi \models^? x{\leq}y \quad \text{iff} \quad \varphi \models \mathrm{pref}^{\bot}_{\pi}(y) \to \mathrm{pref}^{\bot}_{\pi}(x)$$

We define a variable assignment α to be l-safe or r-safe for $\alpha \models^? x{\leq}y$ by replacing φ literally with α in the above definitions. Note that our safety notions depend on the chosen structure of trees over which we interpret our formulas.

We first illustrate these concepts by a judgment with a unary function symbol:

$$z{=}\top \wedge f(z){\leq}y \models^? x{\leq}y \qquad \text{(no)}$$

Here, ε is r-safe but not l-safe. All other paths $\pi \in 1^+$ are both l-safe and r-safe. There is a variable assignment α which contradicts entailment: $\alpha(x) = \top$, $\alpha(z) = \top, \alpha(y) = f(\top)$. This shows that ε is indeed not l-safe for $\alpha \models^? x{\leq}y$.

Proposition 6.1
Entailment $\varphi \models x{\leq}y$ holds if and only if all words $\pi \in \{1, \ldots, \mathit{arity}(f)\}^*$ are l-safe and r-safe for $\varphi \models^? x{\leq}y$.

PROOF. We first assume that entailment does not hold and show that either l-safety or r-safety can be contradicted for some path. As argued above, there exists an unsafe path π such that $\varphi \wedge y(\pi) <_\Sigma x(\pi)$ is satisfiable. Let α be a solution of this formula.

1. If $\alpha(y)(\pi) =_\Sigma \bot$ then $\alpha \models \mathrm{pref}^{\bot}_{\pi}(y)$. Since $\alpha \models y(\pi) <_\Sigma x(\pi)$ it holds $\alpha(x)(\pi) \in \{f, \top\}$ which implies $\alpha \models \neg\mathrm{pref}^{\bot}_{\pi}(x)$. Thus, π is not r-safe.
2. Otherwise $\alpha(y)(\pi) =_\Sigma f$ which implies $\alpha \models \neg\mathrm{pref}^{\top}_{\pi}(y)$. Further, it holds $\alpha(x)(\pi) =_\Sigma \top$ which implies $\alpha \models \mathrm{pref}^{\top}_{\pi}(x)$. Thus, π is not l-safe.

For the converse, we assume entailment $\varphi \models x{\leq}y$ and show that all paths are l-safe and r-safe for $\varphi \models^? x{\leq}y$. We fix a path π and solution α of φ, and show that π is l-safe and r-safe for $\alpha \models^? x{\leq}y$. Let π' be the longest prefix of π which belongs to $D_{\alpha(x)} \cap D_{\alpha(y)}$.

1. If $\alpha(x)(\pi') =_\Sigma \bot$ then $\alpha \models \neg\mathrm{pref}^{\top}_{\pi}(x)$ so that π is l-safe for $\alpha \models^? x{\leq}y$, and also $\alpha \models \mathrm{pref}^{\bot}_{\pi}(x)$ such that π is r-safe for $\alpha \models^? x{\leq}y$.
2. Suppose $\alpha(x)(\pi') =_\Sigma \top$. Since $\alpha \models \varphi$ and $\varphi \models x{\leq}y$, we know that $\alpha \models x{\leq}y$. Since π' is a node of both trees it follows that $\alpha(x)(\pi') \leq_\Sigma \alpha(y)(\pi')$ and thus $\alpha(y)(\pi') =_\Sigma \top$. Since $\pi' \leq \pi$, $\alpha \models \mathrm{pref}^{\top}_{\pi}(y) \wedge \neg\mathrm{pref}^{\bot}_{\pi}(y)$. Thus, π is l-safe and r-safe for $\alpha \models^? x{\leq}y$.

3. The last possibility is $\alpha(x)(\pi') =_\Sigma f$. We can infer from entailment that $\alpha(y)(\pi') \in \{f, \top\}$. If $\alpha(y)(\pi') =_\Sigma \top$ we are done as before. Otherwise, $\alpha(y)(\pi') =_\Sigma \alpha(x)(\pi') =_\Sigma f$ such that the maximality of π' and $arity(f) \geq 1$ yields $\pi = \pi'$. Now, $\alpha \models \neg\mathrm{pref}_\pi^\bot(y)$ so that π is r-safe, and also $\alpha \models \neg\mathrm{pref}_\pi^\top(x)$ such that π is l-safe for $\alpha \models^? x{\leq}y$.

EXAMPLE. The surprising effect of Example 2.5 seems to go away if one replaces the unary function symbol there by a binary function symbol:

$$x{\leq}f(y,y) \wedge f(x,x){\leq}y \models^? x{\leq}y \qquad \text{(no)}$$

Now, all words in $1^* \cup 2^*$ are l-safe and r-safe, but 12 is neither. Entailment can be contradicted by variable assignments mapping x to $f(f(\bot,\top),\bot)$ and y to $f(f(\top,\bot),\top)$.

EXAMPLE. This example is a little more complicated. Its purpose is to show that entailment in the binary case can also be raised by a similar effect as in Example 2.5. How to understand this effect in general will be explained in Chapter 7.

$$x{\leq}f(y,y) \wedge f(z,z){\leq}y \wedge f(u,u){\leq}z \wedge u{=}\top \models^? x{\leq}y \qquad \text{(yes)}$$

6.2 Cap automata and cap sets

We need a notion of automata that can recognize cap sets. Therefore, we restrict the class of P-automata introduced in [NP99] to the class of so called *cap automata*[1]. We then show that the class of languages recognized by cap automata is precisely the class of cap sets, i.e. those sets of words described by cap set expressions.

A *finite automaton* $\mathcal{A}$ over alphabet A consists of a set Q of *states*, a set $I \subseteq Q$ of *initial* states, a set $F \subseteq Q$ of *final* states, and a set $\Delta \subseteq Q \times (A \cup \{\varepsilon\}) \times Q$ of *transitions*. Note that Δ permits ε transitions and single letter transitions. We will write $\mathcal{A} \vdash q$ if $q \in Q$ is a state of $\mathcal{A}$, $\mathcal{A} \vdash \underline{q}$ if $q \in F$ is a final state of $\mathcal{A}$, and $\mathcal{A} \vdash {>}q$ if $q \in I$ is an initial state of $\mathcal{A}$. The statement $\mathcal{A} \vdash q \xrightarrow{\pi} q'$ says that $\mathcal{A}$ started at q permits a sequence of transitions consuming π and ending in q'. Note that $\mathcal{A} \vdash q \xrightarrow{\varepsilon} q$

[1] Cap automata are the same objects as P-automata, i.e. finite automata with a set of P-edges. The difference between both concepts concerns only the corresponding language definitions. Both definitions coincide for those automata $\mathcal{P}$ that satisfy the following condition (the proof is straightforward): if $\mathcal{P} \vdash q_1 \xrightarrow{\pi} q_2 \xrightarrow{\mu} q_3 \dashrightarrow q_1$ then q_2 is a final state in $\mathcal{P}$. This condition can be assumed w.l.o.g for all cap automata, since it is satisfied by all those constructed in the proof of Proposition 6.2. Thus, cap automata are properly subsumed by the P-automata.

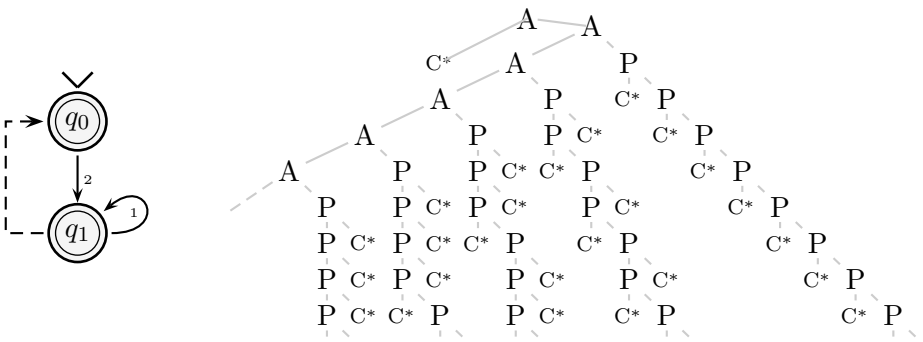


Figure 6.1: A cap automaton with a non context-free language $(21^*)^\circ$.

holds for all states $q \in Q$. We call $\mathcal{A}$ *complete* if for every word $\pi \in A^*$ there exists states q_0 and q_1 such that $\mathcal{A} \vdash \gtrdot q_0 \xrightarrow{\pi} q_1$.

Definition 6.1
A cap automaton $\mathcal{P}$ *over alphabet* A *consists of a finite automaton* $\mathcal{A}$ *over* A *and a set of* P-edges $P \subseteq Q \times Q$. *We write* $\mathcal{P} \vdash q \dashrightarrow q'$ *if* $\mathcal{P}$ *has a P-edge* $(q, q') \in P$. *A cap automaton* $\mathcal{P}$ *over* A *recognizes the following language* $\mathcal{L}(\mathcal{P}) \subseteq A^*$*:*

$$\mathcal{L}(\mathcal{P}) = \{\pi \mid \mathcal{P} \vdash \gtrdot q_0 \xrightarrow{\pi} \underline{\underline{q_1}}\} \cup \{\pi\mu' \mid \mu' \in pr(\mu^*), \mathcal{P} \vdash \gtrdot q_0 \xrightarrow{\pi} \underline{\underline{q_1}} \xrightarrow{\mu} q_2 \dashrightarrow \underline{\underline{q_1}}\}$$

The first set is the language of the finite automaton underlying $\mathcal{P}$. The second set adds the contribution of P-edges: if a cap automaton traverses a P-edge $\mathcal{P} \vdash q_2 \dashrightarrow q_1$ then it must have reached q_2 from q_1 of some word μ, i.e. $\mathcal{P} \vdash q_1 \xrightarrow{\mu} q_2 \dashrightarrow q_1$; in the sequel the automaton can loop through μ^* and quit the loop at any time.

Fig. 6.1 contains a cap automaton over the alphabet $\{1, 2\}$ that recognizes the non-context free cap set from the introduction, i.e. described by the cap set expression $(21^*)^\circ$. We generally draw cap automata as one draws finite automata but with additional dashed arrows to indicate P-edges.

The tree on the right in Fig. 6.1 represents the language recognized by this cap automaton. The language of a cap automaton $\mathcal{P}$ with alphabet $\{1, \ldots, n\}$ is drawn as a n-ary *c*lass tree. This is a complete infinite n-ary tree whose nodes are labeled by classes A, P, and C. Each node of the class tree is a word in $\{1, \ldots, n\}^*$ that is labeled by the class that $\mathcal{P}$ adjoins to it. We assign the class C to all words in the complement of $\mathcal{L}(\mathcal{P})$ of a cap automaton $\mathcal{P}$. The words with class A are recognized

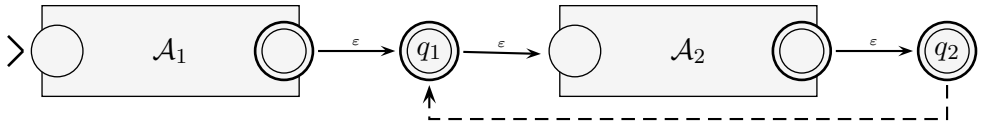


Figure 6.2: Construction of a cap automaton for the language $\mathcal{L}(\mathcal{A}_1)\mathcal{L}(A_2)^\circ$.

by the finite automaton underlying $\mathcal{P}$. All remaining words belong to class P. These are accepted by $\mathcal{P}$ but not by the underlying finite automaton.

We now explain the name *cap*: it is an abbreviation for the regular expression $(\,\mathrm{C} \cup \mathrm{A}^+\mathrm{P}^*\,)^*$. Branches in class trees of cap automata always satisfy that expression. This means that all nodes of class P in a class tree have a mother node in either of the classes A or P. To see this, note first that root nodes of class trees can never belong to class P. Thus, all P nodes must have a mother. Furthermore, the mother of a P node cannot belong to the C class due to the cap property.

Proposition 6.2
Cap set expressions and cap automata recognize precisely the same class of languages. Universality of cap set expressions and cap automata are equivalent modulo deterministic polynomial time transformations.

PROOF. For the one direction, let R_{q_1,q_2} be a regular expression for the set $\{\pi \mid \mathcal{P} \vdash q_1 \xrightarrow{\pi} q_2\}$ then the language of a cap automaton is equal to the union of $\cup_{\mathcal{P}\vdash\, >q_0} \cup_{\mathcal{P}\vdash \underline{\underline{q_1}}} R_{q_0,q_1}$ and $\cup_{\mathcal{P}\vdash\, >q_0} \cup_{\mathcal{P}\vdash \underline{\underline{q_1}}} \cup_{\mathcal{P}\vdash q_2 \dashrightarrow q_1} R_{q_0,q_1}(R_{q_1,q_2})^\circ$. The needed regular expressions can be computed in polynomial time

For the converse, we first note that the class of languages recognized by cap automata is closed under union since cap automata may have several initial states. There thus only remains to build cap automata for expressions $R_1R_2^\circ$. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be finite automata that recognize R_1 respectively R_2. W.l.o.g. we can assume that both automata have a unique initial and a unique final state. Multiple initial or final states of finite automata (but not of cap automata) can be eliminated by introducing new ε-transitions. We now compose $\mathcal{A}_1$ and $\mathcal{A}_2$ into a new cap automaton that recognizes the language of $R_1R_2^\circ$ as illustrated in Fig. 6.2: we add two fresh final states q_1 and q_2 and link $\mathcal{A}_1$ and $\mathcal{A}_2$ over these states. This requires 3 new ε-edges and a new P-edge from q_2 to q_1. To account for the prefix closure within the $^\circ$ operator, we finally turn all states of $\mathcal{A}_2$ into additional final states.

7 Automata Construction

We present a construction for cap automata that can test l-safety and r-safety for entailment judgments. The same construction applies for the three structures of finite, regular, and possibly infinite trees. The only difference is hidden in a subroutine for testing satisfiability (see Chapter 3).

7.1 Left and right automata

As a prerequisite for our automata construction, we use the closure algorithm of Chapter 3. This algorithm computes a set of inequalities of the form $x \leq y$ that are syntactically supported by a constraint φ.

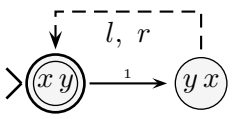
The automata construction is given in Table 7.1. For each entailment judgment $\varphi \models^? x \leq y$ we construct a left automaton $\mathcal{P}_l(\varphi \models^? x \leq y)$ and a right automaton $\mathcal{P}_r(\varphi \models^? x \leq y)$. The left automaton is supposed to accept all l-safe paths for $\varphi \models^? x \leq y$, and the right automaton all r-safe paths (up to appropriate assumptions). Entailment then holds if and only if the languages of both cap automata are universal. Note that it remains open whether the set of simultaneously l-safe and r-safe paths can be recognized by a single cap automaton. The problem is that cap automata are not closed under intersection (proof omitted).

The left and right automata always have the same states, transitions, and initial states. When testing for $\varphi \models^? x \leq y$ the only **initial state** is (x, y). A state (u, s) of the left automaton is made **final** if there is an upper bound $u \leq f(u_1, \ldots, u_n)$ in φ, which proves that the actual path is l-safe. The **descend** rule can also be applied in that case. The safety check then continues in some state (u_i, s') and extends the actual path by i. It can chose $s' = \_$ while ignoring the right hand side, or if s is also a variable descend simultaneously on the right hand side. There are three rules that prove that the actual path and **all** its extensions are l-safe: **bot**, **top**, and **reflexivity**. Finally there is a single rule that adds **P-edges** to the left automaton. The rules for the right automaton are symmetric.

When drawing the constructed left and right automata (Fig. 7.1 and 7.2), we always share the states and transitions for reasons of economy. Different elements of the two

alphabet	$A_\Sigma = \{1, \ldots, arity(f)\}$	
states	$\mathcal{P}_\theta \vdash (s, s')$	if $s, s' \in V(\varphi) \cup \{x, y, \_\}$
	$\mathcal{P}_\theta \vdash \underline{\underline{all}}$	
initial state	$\mathcal{P}_\theta \vdash \ >(x, y)$	
final states	$\mathcal{P}_l \vdash \underline{\underline{(u, s)}}$	if $\varphi \vdash u \leq i(u'),\ i \in A_\Sigma$
	$\mathcal{P}_r \vdash \underline{\underline{(s, v)}}$	if $\varphi \vdash i(v') \leq v,\ i \in A_\Sigma$
descend	$\mathcal{P}_\theta \vdash (u, s) \xrightarrow{i} (u', \_)$	if $\varphi \vdash u \leq i(u'),\ i \in A_\Sigma$
	$\mathcal{P}_\theta \vdash (s, v) \xrightarrow{i} (\_, v')$	if $\varphi \vdash i(v') \leq v,\ i \in A_\Sigma$
	$\mathcal{P}_\theta \vdash (u, v) \xrightarrow{i} (u', v')$	if $\varphi \vdash u \leq i(u'),\ \varphi \vdash i(v') \leq v,\ i \in A_\Sigma$
bot	$\mathcal{P}_\theta \vdash \underline{\underline{(u, s)}} \xrightarrow{i} \underline{\underline{all}}$	if $\varphi \vdash u \leq u',\ u' = \bot$ in $\varphi,\ i \in A_\Sigma$
top	$\mathcal{P}_\theta \vdash \underline{\underline{(s, v)}} \xrightarrow{i} \underline{\underline{all}}$	if $\varphi \vdash v' \leq v,\ v' = \top$ in $\varphi,\ i \in A_\Sigma$
reflexivity	$\mathcal{P}_\theta \vdash \underline{\underline{(u, v)}} \xrightarrow{i} \underline{\underline{all}}$	if $\varphi \vdash u \leq v,\ i \in A_\Sigma$
all	$\mathcal{P}_\theta \vdash \underline{\underline{all}} \xrightarrow{i} \underline{\underline{all}}$	if $i \in A_\Sigma$
P−edges	$\mathcal{P}_l \vdash (u, s) \dashrightarrow (v, u)$	
	$\mathcal{P}_r \vdash (s, v) \dashrightarrow (v, u)$	

Table 7.1: Construction of the cap automata $\mathcal{P}_\theta = \mathcal{P}_\theta(\varphi \models^? x \leq y)$ for both sides $\theta \in \{l, r\}$.



$x \leq f(y) \wedge f(x) \leq y \models^? x \leq y$

Figure 7.1: Automata construction for Example 2.5. Entailment holds.

automata carry extra annotations. Final states of the left (right) automaton are put into a left (right) double circle. If a state is final for both automata then it is drawn within a complete double circle. We annotate P-edges of the left automaton by l and of the right automaton with r.

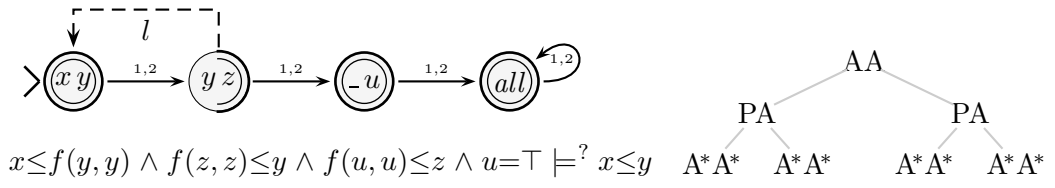


Figure 7.2: Automata construction for Example 6.1. Entailment holds.

7.2 Examples

We first illustrate the automata construction for the unary Example 2.5, recalled in Fig. 7.1. The alphabet of both automata is the singleton $\{1\}$. The relevant states are $\{(x, y), (y, x)\}$; all others are either unreachable or do not lead to a final state. The constraints $x{\leq}f(y)$ and $f(x){\leq}y$ let both cap automata **descend** simultaneously by the transition $(x, y) \xrightarrow{1} (y, x)$ and turn (x, y) into a **final state** of both automata. There are **P-edges** $(y, x) \dashrightarrow (x, y)$ for both cap automata. Note that we ignore the symmetric P-edges $(x, y) \dashrightarrow (y, x)$ in the picture since they don't contribute to the respective languages.

Fig. 7.1 also contains the class trees for both cap automata but in an overlaid fashion. The languages of both cap automata are universal due to their P-edges. Given that our construction is sound (see Sec. 7.4) this proves entailment.

We now consider the more complex binary Example 6.1 in Fig. 7.2 where the alphabet is $\{1, 2\}$. The constraint $f(u, u){\leq}z$ permits to **descend** from (y, z) while ignoring the variable y on the left hand side; this justifies the transition $(y, z) \xrightarrow{1,2} (\_, u)$. Since the left hand side is ignored, the state (y, z) is only put to the **final states** of the right automaton. The **top** rule can be applied to $u{=}\top$; hence there are transitions $(\_, u) \xrightarrow{1,2} \mathit{all}$ where $(\_, u)$ and all are universal states according to the **all** rule. Finally, there is a **P-edge** $(y, z) \dashrightarrow (x, y)$ for the left cap automaton. We again ignore the symmetric **P-edge** $(x, y) \dashrightarrow (y, z)$ since it does not contribute to the language. The languages of both automata are again universal, in case of the left automaton because of a P-edge.

The next example shows that decomposition closure as provided by the notion of syntactic support is needed for completeness. We consider

$$f(x){\leq}z \wedge z{\leq}f(z) \wedge f(z){\leq}y \models^? x{\leq}y \qquad \text{(yes)}$$

Let φ be the left hand side. Since φ contains $f(x){\leq}z \wedge z{\leq}f(z)$, it supports $\varphi \vdash x{\leq}z$ syntactically. Thus, $\varphi \models x{\leq}f(z){\leq}y$ so that entailment holds. This can also be proved

through our automaton construction. First note that $\varphi \vdash x{\leq}1(z)$ holds (since $\varphi \vdash x{\leq}z$ and $z{\leq}f(z)$ in φ). Furthermore, $f(z){\leq}y$ in φ so that $\varphi \vdash 1(z){\leq}y$. Hence, we can **descend** to the first child simultaneously for x and y with the transition:

$$\mathcal{P}_\theta(\varphi \models^? x{\leq}y) \vdash \underline{\underline{(x,y)}} \xrightarrow{1} (z,z)$$

This applies for both sides $\theta \in \{l,r\}$ and proves that ε is l-safe and r-safe for $\varphi \models^? x{\leq}y$. **Reflexivity** shows that all words in 1^+ are l-safe and r-safe too. Thus, our automata construction proves entailment to hold as well.

7.3 Result

The automata construction is sound and complete for both cases, the structures of finite resp. possibly infinite trees.

Proposition 7.1 (Soundness and Completeness)
Let $\theta \in \{l,r\}$ a side, φ be a constraint and $x,y \in V(\varphi)$ variables. If φ is satisfiable then $\mathcal{P}_\theta(\varphi \models^? x{\leq}y)$ accepts the set of all those paths that are θ-safe for $\varphi \models^? x{\leq}y$.

Soundness will be proved in Section 7.4 (Proposition 7.2) and completeness in Section 7.5 (Proposition 7.5).

Lemma 7.1
The automata construction for judgments $\varphi \models^? x{\leq}y$ can be performed in deterministic polynomial time in the size of φ.

PROOF. The closure algorithm can compute all valid judgments of the form $\varphi \vdash u{\leq}v$ where u,v are variables in $V(\varphi)$ in time $O(m^3)$ and store its result in a $O(m^2)$ table where m is the size of φ. The left and right automata have $O(m^2)$ many states. We have to show that we can apply all construction rules in polynomial time. This is non-obvious for the three **descend** rules, at least not at first sight. But note that the set

$$\{z \mid \varphi \vdash x{\leq}i(z)\}$$

can be computed in polynomial time for a given x and $1 \leq i \leq n = \mathit{arity}(f)$: First, one computes all variables x' such that $\varphi \vdash x{\leq}x'$ in time $O(m)$. Second, one loops over all such x' while computing all y_i for which there exists some literal $x'{\leq}f(y_1,\ldots,y_n)$ in φ. This requires time $O(m)$ for each individual x'. Finally one loops for all such y_i while computing all z such that $\varphi \vdash y_i{\leq}z$. This can be done in time $O(m)$ again. Thus, we obtain an $O(m^3)$ algorithm to compute the above set for a given x and i.

This set $\{z \mid \varphi \vdash x{\leq}i(z)\}$ has to be computed for all $x \in V(\varphi)$ and $1 \leq i \leq n$, i.e., for $O(m^2)$ many pairs. Thus, the overall automata construction requires time at most $O(m^6)$ in the size of φ. (We did not try to improve on this upper bound.)

THEOREM 7.1 (Reduction)
NSSE for a signature $\{\bot, f, \top\}$ can be reduced in deterministic polynomial time to the universality problem of cap automata over the alphabet $\{1, \ldots, arity(f)\}$. This holds equally for finite, regular, or possibly infinite trees.

PROOF. The reduction works as follows. Given an entailment judgment $\varphi \models^? x{\leq}y$ we first test whether φ is satisfiable. This tests can be done in polynomial time as we have proved in Theorem 3.1 of Chapter 3. Note that this satisfiability test is the single step of the reduction that differs for finite, resp. regular, or possibly infinite trees.

If φ is unsatisfiable, then entailment holds. Otherwise we construct the left and right automata for $\varphi \models^? x{\leq}y$. This requires at most polynomial time according to Lemma 7.1. Entailment now holds if and only if the languages of both constructed automata are universal as stated by Propositions 7.1 and 6.1.

7.4 Soundness

In this section, we prove the soundness of the automata construction. The proof is non-trivial and requires a new argument compared to [NP99]. This argument (see the proof of Proposition 7.4) is based on Lemma 5.1 from Chapter 5.

Proposition 7.2 (Soundness)
For all φ, variables x, y, and sides $\theta \in \{l, r\}$ it holds that all paths accepted by $\mathcal{P}_\theta(\varphi \models^? x{\leq}y)$ are θ-safe for $\varphi \models^? x{\leq}y$.

We only consider the left side $\theta = l$. We proceed in two steps: we first treat accepted words in class A (Proposition 7.3) and second in class P (Proposition 7.4). For both steps, we have to characterize transitions of the constructed automata.

Lemma 7.2 (Transitions without *all* states)
For all constraints φ, variables x, y, u, v, sides $\theta \in \{l, r\}$, and nonempty words $\pi \in \{1, \ldots, arity(f)\}^+$:

1. *$\mathcal{P}_\theta(\varphi \models^? x{\leq}y) \vdash (u, s) \xrightarrow{\pi} (v, s')$ for some s, s' if and only if $\varphi \vdash u{\leq}\pi(v)$.*

2. *$\mathcal{P}_\theta(\varphi \models^? x{\leq}y) \vdash (s, u) \xrightarrow{\pi} (s', v)$ for some s, s' if and only if $\varphi \vdash \pi(v){\leq}u$.*

This lemma would fail for $\pi = \varepsilon$. But this does not matter since if $\varphi \vdash u{\leq}v$ then **reflexivity** yields for all non-empty words π:

$$\mathcal{P}_\theta(\varphi \models^? x{\leq}y) \vdash \underline{\underline{(u,v)}} \xrightarrow{\pi} \underline{\underline{all}}$$

PROOF. We only prove the first statement for the upper bounds for the implication from right to left. (The second property is analogous but for lower bounds.) The proof is by induction on the length of $\pi \neq \varepsilon$.

First, assume $\pi = i$ is a single letter path where $i \in \mathcal{A}_\Sigma$. The supported upper bound $\varphi \vdash u{\leq}i(v)$ permits to apply the **descend** rule. Hence $\mathcal{P}_\theta(\varphi \models^? x{\leq}y) \vdash (u,\_) \xrightarrow{i} (v,\_)$ for arbitrary $\theta \in \{l,r\}$.

Second, consider a path π of length at least two. We can decompose π into $\pi = \pi' i$ for some nonempty path π' and $i \in \mathcal{A}_\Sigma$. Also we can decompose $\varphi \vdash u{\leq}\pi(v)$ into $\varphi \vdash u{\leq}\pi'(v')$ and $\varphi \vdash v'{\leq}i(v)$ for some variable v'. The induction hypothesis applies twice and yields $\mathcal{P}_\theta(\varphi \models^? x{\leq}y) \vdash (u,\_) \xrightarrow{\pi} (v',\_)$ and $\vdash (v',\_) \xrightarrow{i} (v,\_)$.

Lemma 7.3 (Bounds and safety)

1. *If $\alpha \models x{\leq}\pi(u)$ then all proper prefixes of π are l-safe for $\alpha \models^? x{\leq}y$.*
2. *If $\alpha \models x{\leq}\pi(u) \wedge u{=}\bot$ then all paths $\pi\pi'$ with $\pi' \in A_\Sigma^*$ are l-safe for $\alpha \models^? x{\leq}y$.*
3. *If $\alpha \models x{\leq}\pi(u) \wedge \pi(u){\leq}y$ then all paths $\pi\pi'$ with $\pi' \in A_\Sigma^*$ are l-safe for $\alpha \models^? x{\leq}y$.*

PROOF. We only prove 1 (the other two cases are similar). Let π' be a proper prefix of π. Every solution $\alpha \models x{\leq}\pi(u)$ satisfies either $\alpha(x)(\pi') =_\Sigma f$ or there exists a path $\nu < \pi'$ with $\alpha(x)(\nu) =_\Sigma \bot$; thus all π' are l-safe for $\alpha \models^? x{\leq}y$.

Proposition 7.3 (Soundness for class A)
For all φ, variables x, y it holds that all paths π with class A *accepted by $\mathcal{P}_l(\varphi \models^? x{\leq}y)$ are l-safe for $\varphi \models^? x{\leq}y$.*

PROOF. We have to consider all recognizing transitions of the constructed finite automaton. According to the automaton construction there are three possibilities for doing this.

1. Assume that the path is accepted in a state to which the **final states** rule applies, i.e. π is recognized by a transition of the following form:

$$\mathcal{P}_l(\varphi \models^? x{\leq}y) \vdash \ {>}(x,y) \xrightarrow{\pi} \underline{\underline{(u,s)}} \xrightarrow{i} (u',s').$$

Lemma 7.2 yields $\varphi \models x{\leq}\pi i(u')$. Thus π is l-safe for $\varphi \models^? x{\leq}y$ by Lemma 7.3 case 1.

2. Assume π is accepted by the **reflexivity** rule. Then, π must be of the form $\pi_1\pi_2$, where we can identify transition of the following form where $\varphi \vdash u{\leq}u'$:

$$\mathcal{P}_l(\varphi \models^? x{\leq}y) \vdash \ \gtrdot(x,y) \xrightarrow{\pi_1} \underline{\underline{(u,u')}}.$$

Lemma 7.2 yields $\varphi \models x{\leq}\pi_1(u)$. By symmetrical reasoning $\varphi \models \pi_1(u'){\leq}y$ and thus $\varphi \models \pi_1(u){\leq}y$. Case 3 of Lemma 7.3 shows that π is l-safe for $\varphi \models^? x{\leq}y$.

3. Assume π is accepted by the **bot** rule (the case that the **top** rule fires is analogous). Then, π must be of the form $\pi_1\pi_2$, where we can identify transition of the following form:

$$\mathcal{P}_l(\varphi \models^? x{\leq}y) \vdash \ \gtrdot(x,y) \xrightarrow{\pi_1} \underline{\underline{(u,s)}},\ \varphi \vdash u{\leq}u',\ \text{and } u'{=}\bot \text{ in } \varphi.$$

Again Lemma 7.2 yields $\varphi \models x{\leq}\pi_1(u')$. Therefore π is l-safe for $\varphi \models^? x{\leq}y$ by Lemma 7.3 case 2.

We now approach the soundness of P-edges. It mainly relies on Lemma 7.4 in combination with Lemma 5.1 on word equations.

Lemma 7.4 (Safety and word equations)
Let $\pi \neq \varepsilon$ be a path, u, v variables, and $\alpha \models u{\leq}\pi(v)$. All words π' with $\pi' \in pr(\pi\pi')$ are l-safe for $\alpha \models^? u{\leq}v$.

PROOF. We distinguish whether π' belongs to $D_{\alpha(v)}$ or not.

a. Case $\pi' \in D_{\alpha(v)}$. It follows in this case from $\alpha \models u{\leq}\pi(v)$, that $\alpha \models \exists v'(u{\leq}\pi\pi'(v'))$. By Lemma 7.3 all proper prefixes of $\pi\pi'$ are l-safe for $\alpha \models^? u{\leq}v$. Thus π' has this property since π' is a prefix of $\pi\pi'$ and $\pi \neq \varepsilon$ by assumption.

b. Case $\pi' \notin D_{\alpha(v)}$. Let π'' be the maximal prefix of π' in $D_{\alpha(v)}$. Hence, $\alpha(v)(\pi'') \in \{\bot, \top\}$. First we assume the case $\alpha(v)(\pi'') =_\Sigma \top$ which implies that all paths σ with $\pi'' \leq \sigma$, in particular π', are l-safe. Second we assume the left case $\alpha(v)(\pi'') =_\Sigma \bot$. Since $\alpha \models u{\leq}\pi(v)$, there exists a path π''' with $\pi''' \leq \pi\pi'$ such that $\alpha(u)(\pi''') =_\Sigma \bot$. Both together, $\pi''' \leq \pi\pi'$ and the assumption $\pi' \leq \pi\pi'$ show that the paths π''' and π' are comparable: if $\pi' \geq \pi'''$ then π' is l-safe according to the definition of l-safe. Otherwise, $\pi' < \pi'''$ holds and $\alpha(u)(\pi') =_\Sigma f$ implies π' to be l-safe.

Lemma 7.5 (Composing safety)
If $\alpha \models x{\leq}\pi(u)$, $\alpha \models \pi(v){\leq}y$, and π' is l-safe for $\alpha \models^? u{\leq}v$ then $\pi\pi'$ is l-safe for $\alpha \models^? x{\leq}y$.

PROOF. It follows from the assumption π' is l-safe for $\alpha \models^? u{\leq}v$ that $\alpha(u)(\pi') =_\Sigma f$ or that there exists $\pi''{\leq}\pi'$ with $\alpha(u)(\pi'') =_\Sigma \bot$ or $\alpha(v)(\pi'') =_\Sigma \top$. The assumption $\alpha \models x{\leq}\pi(u)$ and $\alpha \models \pi(v){\leq}y$ imply that $\alpha(x)(\pi\pi') =_\Sigma f$ or there exists $\pi''{\leq}\pi\pi'$ with $\alpha(x)(\pi'') =_\Sigma \bot$ or $\alpha(y)(\pi'') =_\Sigma \top$; thus $\pi\pi'$ is l-safe for $\alpha \models^? x{\leq}y$.

Proposition 7.4 (Soundness for class P)
For all φ and variables x, y, all paths of class P *accepted by $\mathcal{P}_l(\varphi \models^? x{\leq}y)$ are l-safe for $\varphi \models^? x{\leq}y$.*

PROOF. A path ν of class P can only be recognized by using a P-edge. Thus, there exist words μ, μ', π such that $\nu = \pi\mu'$, $\mu' \in pr(\mu^*)$ and for some $u, v \in V(\varphi) \cup \{x, y\}$ and $s \in V(\varphi) \cup \{x, y, \_\}$:

$$\mathcal{P}_l(\varphi \models^? x{\leq}y) \vdash (x, y) \xrightarrow{\pi} \underline{\underline{(u, v)}} \xrightarrow{\mu} (v, s) \dashrightarrow \underline{\underline{(u, v)}}$$

Lemma 7.2 yields $\varphi \models x{\leq}\pi(u)$, $\varphi \models \pi(v){\leq}y$, and $\varphi \models u{\leq}\mu(v)$. We fix an arbitrary solution $\alpha \models \varphi$ and show that ν is l-safe for $\alpha \models x{\leq}y$. Note that $\mu \neq \varepsilon$ since ν would belong to class A otherwise. We can thus apply **Lemma 5.1 on word equations** to our assumption $\mu' \in pr(\mu^*)$ to derive $\mu' \in pr(\mu\mu')$. This verifies the assumptions of Lemma 7.4 which shows that μ' is l-safe for $\alpha \models^? u{\leq}v$. Finally, the composition Lemma 7.5 shows that $\pi\mu'$ is l-safe for $\alpha \models^? x{\leq}y$ as required.

7.5 Completeness

We prove the completeness of the automata construction. The proof is simplified in many aspects compared to its relatives [NP99].

Proposition 7.5 (Completeness)
Let φ be a constraint with variables $x, y \in V(\varphi)$ and $\theta \in \{l, r\}$ a side. If φ is satisfiable (for finite, regular, resp. possibly infinite trees) then $\mathcal{P}_\theta(\varphi \models^? x{\leq}y)$ accepts all paths that are θ-safe for $\varphi \models^? x{\leq}y$ (with respect to the considered structure of finite, regular, resp. possibly infinite trees).

In a first step, we reduce Proposition 7.5 to simpler statements in Proposition 7.6 and 7.7. By symmetry, we can restrict ourselves to the case of r-safety. This notion can be reformulated on basis of standard logical transformations.

Lemma 7.6
Let ν be a word in $\{1, \ldots arity(f)\}^*$ and $\varphi \models^? x{\leq}y$ and entailment judgment. Then ν is not r-safe for $\varphi \models^? x{\leq}y$ if and only if $\varphi \wedge \mathrm{pref}^\bot_\nu(y) \wedge \neg\mathrm{pref}^\bot_\nu(x)$ is satisfiable.

PROOF. The word ν is r-safe for $\varphi \models^? x{\leq}y$ iff $\varphi \models \mathrm{pref}^{\perp}_{\nu}(y) \rightarrow \mathrm{pref}^{\perp}_{\nu}(x)$ iff $\varphi \wedge \neg(\mathrm{pref}^{\perp}_{\nu}(y) \rightarrow \mathrm{pref}^{\perp}_{\nu}(x))$ is unsatisfiable, i.e., if $\varphi \wedge \mathrm{pref}^{\perp}_{\nu}(y) \wedge \neg\mathrm{pref}^{\perp}_{\nu}(x)$ is satisfiable.

In order to prove Proposition 7.5 we can assume a satisfiable constraint φ with variables x, y, and a path ν in $\{1, \ldots, \mathit{arity}(f)\}^*$ that does not belong to the language of $\mathcal{P}_l(\varphi \models^? x{\leq}y)$. We then have to prove that ν is not r-safe for $\varphi \models^? x{\leq}y$. Using the above Lemma, this is equivalent to the satisfiability of the following formula:

$$\varphi \wedge \mathrm{pref}^{\perp}_{\nu}(y) \wedge \neg\mathrm{pref}^{\perp}_{\nu}(x)$$

We can eliminate the negative subformula $\neg\mathrm{pref}^{\perp}_{\nu}(x)$ by a simple trick. Let

$$x_{\varepsilon} =_{\mathrm{def}} x$$

and fix a set $F_{\nu}(x)$ of fresh and distinct variables x_{μ} for finitely many nonempty paths μ that are successors of prefixes of ν:

$$F_{\nu}(x) =_{\mathrm{def}} \{\ x_{\pi i} \mid \varepsilon \leq \pi \leq \nu,\ \ 1 \leq i \leq \mathit{arity}(f)\ \}$$

Note that $F_{\nu}(x)$ does *not* contain x_{ε}. Next, we define a constraint $\mathrm{low}_{\nu(f)}(x)$ which imposes the lower bound $\nu(f)$ on x:

$$\mathrm{low}_{\nu(f)}(x) \ =_{\mathrm{def}} \bigwedge_{\varepsilon \leq \pi \leq \nu} f(x_{\pi 1}, \ldots, x_{\pi n}) \leq x_{\pi}$$

Lemma 7.7
The constraint $\mathrm{low}_{\nu(f)}(x)$ *is satisfaction equivalent to* $\neg\mathrm{pref}^{\perp}_{\nu}(x)$.

PROOF. Indeed, the existential formula $\exists F_{\nu}(x).\ \mathrm{low}_{\nu(f)}(x)$ is equivalent to $\neg\mathrm{pref}^{\perp}_{\nu}(x)$. Note first that $\mathrm{low}_{\nu(f)}(x) \vdash \nu(f){\leq}x$ and hence, $\exists F_{\nu}(x).\ \mathrm{low}_{\nu(f)}(x) \models \nu(f){\leq}x$. Clearly, the converse holds as well, i.e., both formulas are equivalent. Finally, note that $\nu(f){\leq}x$ is also equivalent to $\neg\mathrm{pref}^{\perp}_{\nu}(x)$.

According to Lemma 7.7, the remaining goal is to prove the satisfiability of the formula: $\varphi \wedge \mathrm{pref}^{\perp}_{\nu}(y) \wedge \mathrm{low}_{\nu(f)}(x)$. The cases of possibly infinite or regular trees will be proved in Proposition 7.6. The existence of finite solutions is then derived from the existence of possibly infinite solutions in Proposition 7.7. Before proving these proposition, we formulate some needed properties of the constraint $\mathrm{low}_{\nu(f)}(x)$ in two technical Lemmas 7.8 and 7.9.

Lemma 7.8
The following properties hold for a constraint φ *with variables* $x, u, v \in V(\varphi)$, *words* $\nu, \pi, \mu \in \{1, \ldots, \mathit{arity}(f)\}^*$ *such that* $x_{\mu} \in F_{\nu}(x)$. *Recall that* $F_{\nu}(x)$ *is a set of fresh variables disjoint from* $V(\varphi)$. *Let* $\varphi' = \varphi \wedge \mathrm{low}_{\nu(f)}(x)$.

(1) $\varphi' \vdash u \leq \pi(v)$ *if and only if* $\varphi \vdash u \leq \pi(v)$.

(2) $\varphi' \vdash x_\mu \leq \pi(v)$ *if and only if* $\varphi \vdash x \leq \mu(x')$, $\varphi \vdash x' \leq \pi(v)$ *for some* x'.

(3) $\varphi' \vdash \pi(u) \leq v$ *if and only if* $\varphi \vdash \pi(u) \leq v$.

(4) $\varphi' \vdash \pi(x_\mu) \leq v$ *if and only if* $\varphi \vdash x \leq \mu'(x')$ *and* $\varphi \vdash \pi'(x') \leq v$ *where* $\mu = \mu'\nu$, $\pi = \pi'\nu$ *for some* μ', π', ν, x'.

(5) $\varphi' \vdash \mu' i(x_{\mu\mu' i}) \leq x_\mu$ *if* $\mu\mu' \leq \nu$ *and* $i \in \{1, \ldots, arity(f)\}$.

(6) $\varphi' \vdash \varepsilon(x_\mu) \leq x_\mu$ *and* $\varphi' \vdash x_\mu \leq \varepsilon(x_\mu)$.

PROOF. The proofs of these properties are tedious but not difficult. Note that (5) and (6) are trivial but they will simplify the proof.

All inverse implications are straightforward so that we only treat the most complicated property (4) explicitly. Let the right hand side of (4) be true. Since $\mu \in V_\nu(X)$ we have $\text{low}_{\nu(f)}(x) \vdash \mu(x_\mu) \leq x$ and thus $\text{low}_{\nu(f)}(x) \vdash \mu'\nu(x_\mu) \leq x$. Because of $\varphi \vdash x \leq \mu'(x')$ we can apply the decomposition Lemma 7.2 which yields $\varphi' \vdash \nu(x_\mu) \leq x'$. Combined with $\varphi \vdash \pi'(x') \leq v$ this implies $\varphi' \vdash \pi'\nu(x_\mu) \leq v$ and hence $\varphi' \vdash \pi(x_\mu) \leq v$ as required.

We prove all remaining implications of (1)–(5) together with (6) simultaneously. For this we define a new set $C_{\varphi'}$ of path constraints: a constraint ψ is in $C_{\varphi'}$ if and only if one of the properties (1)–(6) licenses $\varphi' \vdash \psi$. In the rest of the proof we do not distinguish between $\varepsilon(x) \leq y$ in $C_{\varphi'}$ and $x \leq \varepsilon(y)$ in $C_{\varphi'}$. We also write $x \leq y$ in $C_{\varphi'}$ in each of both cases. It remains to prove that $C_{\varphi'}$ is closed under the conditions given in Table 3.4 of lower and upper bounds. We restricted ourself to prove that $C_{\varphi'}$ is again closed under reflexivity, transitivity, decomposition. The set $C_{\varphi'}$ is closed under reflexivity for all variables in $V(\varphi)$ by property (1) and for all variables in $F_\nu(x)$ by (6). For transitivity and decomposition we have to consider literals of the restricted form $u \leq v$ where u and v are distinct variables. Such literals are defined in (1)–(4) where $\pi = \varepsilon$. They are not defined in (5) or (6).

We prove that $C_{\varphi'}$ is closed under transitivity. There is only one interesting case left where the transitivity rule can be applied. Let $v \leq u$ in $C_{\varphi'}$ with $v, u \in V(\varphi)$ be contributed by $\varphi \vdash v \leq u$ in the case of property (1) or (3). Also let $x_\mu \leq v$ in $C_{\varphi'}$ be contributed by (2) or (4) which require $\varphi \vdash x \leq \mu(x')$, $\varphi \vdash x' \leq v$ for some x, x', v, μ. Following Lemma 3.5 it holds $\varphi \vdash x \leq \mu(u)$ and thus, $x_\mu \leq u$ in $C_{\varphi'}$ again by (2).

We prove $C_{\varphi'}$ to be closed under decomposition.

1. Assume $u \leq f(\ldots, u_i, \ldots) \wedge f(\ldots, v_i, \ldots) \leq v$ in φ for variables $u, u_i, v, v_i \in V(\varphi)$. Also assume $v \leq u$ in $C_{\varphi'}$ by (1) or (3) contributed by $\varphi \vdash v \leq u$. Then, $\varphi \vdash v_i \leq u_i$ and also $v_i \leq u_i$ in $C_{\varphi'}$ by (2).

2. Assume $u{\leq}f(\ldots,u_i,\ldots)$ in φ with $u,u_i \in V(\varphi)$ and $f(x_1,\ldots,x_n) \leq x_\varepsilon$ in $\mathrm{low}_{\nu(f)}(x)$. Note that $x_\varepsilon = x$, $x \in V(\varphi)$ by assumption. Also assume $x{\leq}u$ in $C_{\varphi'}$ which is contributed by (1) or (3) with $\varphi \vdash x{\leq}u$. Then, $x_i{\leq}u_i$ in $C_{\varphi'}$ by (2).

3. Assume $u{\leq}f(\ldots,u_i,\ldots)$ in φ with $u,u_i \in V(\varphi)$ and $f(x_{\pi 1},\ldots,x_{\pi n}) \leq x_\pi$ in $\mathrm{low}_{\nu(f)}(x)$ where $\pi \neq \varepsilon$. Thus, $x_\pi \in F_\nu(x)$. Then, $x_i{\leq}u_i$ in $C_{\varphi'}$ by (2).

Lemma 7.9
Let $\nu \in \{1,\ldots,arity(f)\}^*$ and let φ be a constraint with variables x, y. It holds that $\varphi \wedge \mathrm{low}_{\nu(f)}(x) \vdash \nu(f){\leq}y$ if and only if

1. *$\varphi \vdash \nu(f){\leq}y$, or*
2. *there exist a state z and paths $\pi_1{\leq}\nu$, π_2 such that $\varphi \vdash \pi_1\pi_2(z){\leq}y$, $\varphi \vdash x{\leq}\pi_1(z)$, and if $\pi_2 \neq \varepsilon$ then also $\nu \in \pi_1\, pr(\pi_2^*)$.*

PROOF. From right to left. Clearly, $\varphi \vdash \nu(f){\leq}y$ implies $\varphi \wedge \mathrm{low}_{\nu(f)}(x) \vdash \nu(f){\leq}y$. So let $\varphi \vdash \pi_1\pi_2(z){\leq}y$ and $\varphi \vdash x{\leq}\pi_1(z)$. Let ν' be an arbitrary path and $\nu = \pi_1\nu'$. If $\psi \vdash \pi_1\pi_2(z){\leq}y$, $\psi \vdash x{\leq}\pi_1(z)$, and $\psi \vdash \pi_1\nu'(f){\leq}x$ then $\psi \vdash \pi_1\pi_2\nu'(f){\leq}y$ according to Lemma 3.5. Note that $\mathrm{low}_{\nu(f)}(x) \vdash \nu(f){\leq}x$. Thus, $\varphi \wedge \mathrm{low}_{\nu(f)}(x) \vdash \pi_1\pi_2\nu'(f){\leq}y$ which also implies $\varphi \wedge \mathrm{low}_{\nu(f)}(x) \vdash \pi_1\pi(f){\leq}y$ for every prefix $\pi \leq \pi_2\nu'$. The case $\pi = \nu'$, which was to prove, holds for $\pi_2 = \varepsilon$ or $\nu' \in pr(\pi_2^*)$ according to Lemma 5.1.

From left to right. Let assume $\varphi \wedge \mathrm{low}_{\nu(f)}(x) \vdash \nu(f){\leq}y$, $y \in V(\varphi)$ which is equivalent to $\varphi \wedge \mathrm{low}_{\nu(f)}(x) \vdash \nu(u){\leq}y$ together with either (1) $\varphi \vdash \varepsilon(f){\leq}u$ where $u \in V(\varphi)$ or (2) $\mathrm{low}_{\nu(f)}(x) \vdash \varepsilon(f){\leq}u$ where (2a) $u = x$ or (2b) $u \in F_\nu(x)$. Let us assume case (1). By Lemma 7.8 also $\varphi \vdash \nu(u){\leq}y$ since $y,u \in V(\varphi)$. Our case assumption $\varphi \vdash \varepsilon(f){\leq}u$ implies $\varphi \vdash \nu(f){\leq}y$.

Next, assume case (2a). Again by Lemma 7.8 also $\varphi \vdash \nu(x){\leq}y$ since $y,u \in V(\varphi)$. Then, there exist $\pi_1 = \varepsilon, \pi_2 = \nu, z$ with $\varphi \vdash \pi_1\pi_2(z){\leq}y$ and $\varphi \vdash x{\leq}z$. Also $\nu \in pr(\pi_1^*)$.

At last assume the remaining case (2b). Our assumption $\mathrm{low}_{\nu(f)}(x) \vdash \varepsilon(f){\leq}u$, $u \in F_\nu(x)$ holds in the case $u = x_\pi$ where $\pi \leq \nu$. Lemma 7.8.4 together with our assumption $\varphi \wedge \mathrm{low}_{\nu(f)}(x) \vdash \nu(x_\pi){\leq}y$ implies $\varphi \vdash x{\leq}\pi'(x')$, $\varphi \vdash \nu'(x'){\leq}y$ where $\pi{=}\pi'\nu''$, $\nu{=}\nu'\nu''$ for some ν', ν'', x'. Since $\pi \leq \nu$, we get $\pi'\nu'' \leq \nu = \nu'\nu''$. This implies $\pi' \leq \nu'$. So, let $\nu' = \pi'\pi''$ for some π''. Then we identify $\varphi \vdash \pi'\pi''(x'){\leq}y$ and $\varphi \vdash x{\leq}\pi'(x')$ where $\pi' \leq \nu$. Since $\pi'\nu'' \leq \nu'\nu''$ and $\nu' = \pi'\pi''$ it holds that $\pi'\nu'' \leq \pi'\pi''\nu''$ and thus, $\nu'' \leq \pi''^*$ for $\pi'' \neq \varepsilon$ according to Lemma 5.1. It holds that $\nu = \pi'\pi''\nu'' \in \pi'\ pr(\pi''^*)$ in the case $\pi'' \neq \varepsilon$.

We now return to the main line of the completeness proof, i.e., we show that $\varphi \wedge \mathrm{low}_{\nu(f)}(x) \wedge \mathrm{pref}^{\perp}_{\nu}(y)$ is satisfiable.

Proposition 7.6
Consider the structure of possibly infinite or of regular trees respectively. Let ν be a word in $\{1,\ldots,arity(f)\}^*$ that does not belong to the language of $\mathcal{P}_r(\varphi \models^? x{\leq}y)$. Let φ be a satisfiable constraint with variables $x, y \in V(\varphi)$. Then the least solution of the constraint $\varphi \wedge \mathrm{low}_{\nu(f)}(x)$ exists and satisfies $\mathrm{pref}^{\perp}_{\nu}(y)$ simultaneously.

PROOF. Let $\varphi' =_{\mathrm{def}} \varphi \wedge \mathrm{low}_{\nu(f)}(x)$. In order to show that φ' permits a least solution, we first show that it does not contain any label clash (Theorem 3.1). Assume the contrary, i.e., that φ' contains a label clash. Then there exists variables $u, v \in V(\varphi')$ with $\varphi' \vdash u{\leq}v$ such that φ' requires contradicting label bounds for u and v according to $C1$, $C2$, or $C3$ in Table 3.3. Since $\mathrm{low}_{\nu(f)}(x)$ does not impose any upper label bounds, it follows that $v \in V(\varphi)$. If the lower bound for u belongs already to φ then φ contains a label clash. Hence, $\mathrm{low}_{\nu(f)}(x)$ imposes the lower bound on u. But the only lower bounds that $\mathrm{low}_{\nu(f)}(x)$ imposes are f-bounds for some variables x_π; thus $u = x_\pi$ for some π. Furthermore, lower f-bounds clash only with upper $\perp$-bounds, i.e., we have a label clash of kind $C3$:

$$f(...){\leq}x_\pi \text{ in } \mathrm{low}_{\nu(f)}(x), \quad \varphi' \vdash x_\pi{\leq}v, \quad \text{and} \quad v{=}\perp \text{ in } \varphi.$$

Because of $f(...){\leq}x_\pi$ in $\mathrm{low}_{\nu(f)}(x)$ it follows that $\varepsilon \leq \pi \leq \nu$. We next show that $\varphi \vdash x{\leq}\pi(v)$. If $\pi = \varepsilon$ then $x_\pi \in V(\varphi)$ so that part 1 of Lemma 7.8 yields $\varphi \vdash x{\leq}\pi(v)$. Otherwise, $\pi \neq \varepsilon$ so that $x_\pi \in F_\nu(x)$. Part 2 of Lemma 7.8 yield $\varphi \vdash x{\leq}\pi(v)$ in this case, so it holds in all cases.

Lemma 7.2 implies that the automaton $\mathcal{P}_r(\varphi \models^? x{\leq}y)$ reaches state $(v, \_)$ over word π. Since $v{=}\perp$ in φ, the **bot** rule of the automata construction (Table 7.1) lets $\mathcal{P}_r(\varphi \models^? x{\leq}y)$ accepts all words that π is a prefix of. And $\pi \leq \nu$ so that ν is accepted by the right automaton, in contrast to our assumption.

Recall that the least solution $\mathrm{least}_{\varphi'}$ is regular but possibly infinite. In order to prove $\mathrm{least}_{\varphi'} \models \mathrm{pref}^{\perp}_{\nu}(y)$ we assume the contrary. By definition of $\mathrm{least}_{\varphi'}$ (Section 3.2) the contrary holds if and only if φ' one of the following lower bounds for y: either $\pi(\top)$ for some prefix $\pi \leq \nu$ or $\nu(f)$:

$$\varphi' \vdash \pi(\top){\leq}y \quad \text{or} \quad \varphi' \vdash \nu(f){\leq}y$$

In the first case, there exists some equation $z{=}\top$ in φ' such that $\varphi' \vdash \pi(z){\leq}y$. But $z{=}\top$ cannot belong to $\mathrm{low}_{\nu(f)}(x)$. It thus belongs to φ so that $z \in V(\varphi)$. Part 3 of Lemma 7.8 yields $\varphi \vdash \pi(z){\leq}y$. Lemma 7.2 shows that the automaton $\mathcal{P}_r(\varphi \models^? x{\leq}y)$ reaches state $(\_, z)$ over word π. Because of the **top** rule of the automata construction

(Table 7.1) $\mathcal{P}_r(\varphi \models^? x{\leq}y)$ accepts all words of which π is a prefix, and thus ν, in contrast to our assumption.

The remaining second case $\varphi' \vdash \nu(f){\leq}y$ is the crucial step in this proof. Lemma 7.9 leaves only two possibilities that we distinguish:

1. Case $\varphi \vdash \nu(f){\leq}y$. Hence, there exists z such that $\varphi \vdash \nu(z){\leq}y$ and $f(\ldots) \leq z$ in φ. Lemma 7.2 proves that $\mathcal{P}_r(\varphi \models^? x{\leq}y)$ can reach state $(\_, z)$ over word ν. And this state is final since the **final state** rule applies given $f(\ldots) \leq z$ in φ.

2. In the other case, there exist $\pi_1 \leq \nu$, π_2, and z with $\varphi \vdash \pi_1\pi_2(z){\leq}y$, $\varphi \vdash x{\leq}\pi_1(z)$.

   a) In the case $\pi_1 = \pi_2 = \varepsilon$ it holds that $\varphi \vdash x{\leq}y$. The **initial state** and **reflexivity** prove all words including ν to be in in the language of $\mathcal{P}_r(\varphi \models^? x{\leq}y)$.

   b) Let $\pi_2 = \varepsilon$ but $\pi_1 \neq \varepsilon$. Lemma 7.2 and **initial state** imply

   $$\mathcal{P}_r(\varphi \models^? x{\leq}y) \vdash \ >(x, y) \xrightarrow{\pi_1} (z, z).$$

   Since $\pi_1 \leq \nu$ **reflexivity** proves ν to be in in the language of $\mathcal{P}_r(\varphi \models^? x{\leq}y)$.

   c) Let $\pi_1 \neq \varepsilon$ and also $\pi_1 \neq \varepsilon$. Then, there is also the assumption $\nu \in \pi_1\, pr(\pi_2^*)$. Lemma 7.2 shows for some token s:

   $$\mathcal{P}_r(\varphi \models^? x{\leq}y) \vdash \ >(x, y) \xrightarrow{\pi_1} (z, s) \xrightarrow{\pi_2} (\_, z) \dashrightarrow (z, s)$$

   Again, ν is contained in the language of $\mathcal{P}_r(\varphi \models^? x{\leq}y)$ according to the second P-edge rule.

   d) The remaining case is $\pi_1 = \varepsilon$, but $\pi_2 \neq \varepsilon$. Again, $\nu \in \pi_1\, pr(\pi_2^*)$. Since $\varphi \vdash x{\leq}z$ and $\varphi \vdash \pi_2(z){\leq}y$ it holds also $\varphi \vdash \pi_2(x){\leq}y$. Lemma 7.2 shows:

   $$\mathcal{P}_r(\varphi \models^? x{\leq}y) \vdash \ >(x, y) \xrightarrow{\pi_2} (\_, x) \dashrightarrow (x, y)$$

   Again, ν is contained in the language of $\mathcal{P}_r(\varphi \models^? x{\leq}y)$.

We finally treat the case of finite trees. We reuse satisfiability for the infinite case rather then restarting from scratch. This requires another trick which is hidden in the proof of the next proposition.

Proposition 7.7
Let ν be a word in $\{1, \ldots, arity(f)\}^*$ that does not belong to the language of $\mathcal{P}_r(\varphi \models^? x{\leq}y)$ and φ be a constraint that is satisfiable over finite trees and contains variables $x, y \in V(\varphi)$. Then the constraint $\varphi \wedge \mathrm{low}_{\nu(f)}(x) \wedge \mathrm{pref}_\nu^\perp(y)$ is also satisfiable in the structure of finite trees.

PROOF. In a first step, we express the formula $\mathrm{pref}^{\perp}_{\nu}(y)$ by a satisfaction the equivalent constraint $\mathrm{up}_{\nu(\perp)}(y)$:

$$\mathrm{up}_{\nu(\perp)}(y) \;=_{\mathrm{def}}\; y_\nu{=}\perp \wedge \bigwedge_{\varepsilon\leq\pi<\nu} y_\pi{\leq}f(y_{\pi 1},\ldots,y_{\pi n})$$

where $y_\varepsilon =_{\mathrm{def}} y$ and $F_\nu(y)$ is a set of fresh and distinct variables as before. Since $\varphi \wedge \mathrm{low}_{\nu(f)}(x) \wedge \mathrm{pref}^{\perp}_{\nu}(y)$ is satisfiable over possibly infinite trees by Proposition 7.6, we know that

$$\varphi' \quad =_{\mathrm{def}} \quad \varphi \wedge \mathrm{low}_{\nu(f)}(x) \wedge \mathrm{up}_{\nu(\perp)}(y)$$

is also satisfiable over possibly infinite trees. Now comes the trick: The constraint φ' cannot contain a label clash. (Otherwise it were unsatisfiable over possibly infinite trees by Proposition 3.1.) Furthermore, φ' cannot have a cycle clash, given that φ doesn't (by Proposition 3.4) and since the addition of $\mathrm{low}_{\nu(f)}(x) \wedge \mathrm{up}_{\nu(\perp)}(y)$ leaves this property invariant. Hence, Proposition 3.4 shows that φ' has a finite solution; and the satisfaction equivalent formula $\varphi \wedge \mathrm{low}_{\nu(f)}(x) \wedge \mathrm{pref}^{\perp}_{\nu}(y)$ has a finite solution, too.

7.6 Restrictions of constructed automata

Constructed cap automata satisfy a set of restrictions that we must assume for the back translation in Section 9.

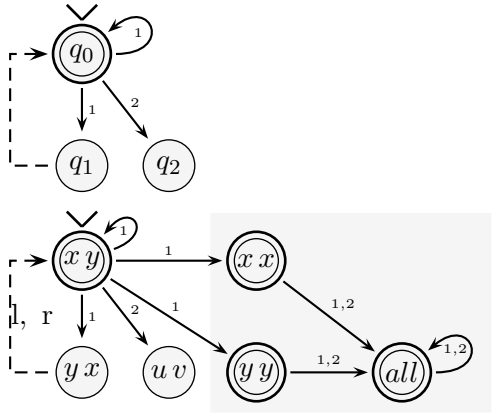
Definition 7.1
We call a cap automaton $\mathcal{P}$ over A restricted *if it is strictly epsilon free, gap universal, strictly cap, and shuffled.*

strictly epsilon free: *$\mathcal{P}$ has a unique initial state and no ε-transition.*

gap universal: *If a final state q_2 can be reached from a non-final state q_1 over some transition $\mathcal{P} \vdash q_1 \xrightarrow{i} \underline{\underline{q_2}}$ with $i \in A$ then q_2 is universal, i.e., for all $\pi \in A^*$ there exists a final state q_3 that can be reach over π from q_2: $\mathcal{P} \vdash \underline{\underline{q_2}} \xrightarrow{\pi} \underline{\underline{q_3}}$.*

strictly cap: *If $\mathcal{P} \vdash q_2 \xrightarrow{\pi} q_3 \dashrightarrow q_1$ with $\pi \neq \varepsilon$ then q_2 is a final state.*

shuffled: *If there are transitions $\mathcal{P} \vdash q \xleftarrow{\pi} q_0 \xrightarrow{\pi} q' \dashrightarrow q$ where $\mathcal{P} \vdash {>}q_0$ is the initial state and $q \neq q'$ then the language $\{\pi' \mid \pi\pi' \in \mathcal{L}(\mathcal{P})\}$ is universal.*



an unshuffled cap automaton,

a suffered extension, and
corresponding entailment judgment

$$\left.\begin{array}{rl} & x{\leq}f(y,u) \wedge f(x,v){\leq}y \\ \wedge & x{\leq}f(x,u) \wedge f(y,v){\leq}y \end{array}\right\} \models^{?} x{\leq}y$$

Figure 7.3: An example for the shuffle property.

We conjecture that these restrictions don't truly restrict the universality problem of cap automata but cannot prove this so far. Indeed, every cap automaton whose underlying finite automaton is deterministic can be made restricted. Again, this is not obvious. The proof exploits that "deterministic" cap automata are always shuffled. But unfortunately, the usual determination procedure fails for cap automata.

However, the shuffle property might be problematic, as illustrated by the example in Table 7.3. On the top, it presents a cap automaton over the alphabet $\{1, 2\}$ which violates the shuffle property at word 1. This automaton rejects all words in $1^*2(1{\cup}2)^*$. Below, a shuffled extension of this automaton is given; the additional nodes are marked in grey. The extended automaton recognizes more words as it only rejects the words in $2(1{\cup}2)^*$ but is still not universal. A corresponding entailment judgment is also given: our automata construction applied to this judgment (for both left and right side) generates the shuffled extension of the original automaton.

The example shows that we cannot simply make an automaton shuffled without extending its language. There are also examples, where the shuffle extension of a non-universal automata becomes universal. So it remains open, whether assuming the shuffle property restricts the universality problem of cap automata or not.

Proposition 7.8
Constructed cap automata $\mathcal{P}_\theta(\varphi \models^{?} x{\leq}y)$ are restricted.

PROOF. Let $\mathcal{P}_l = \mathcal{P}_l(\varphi \models^{?} x{\leq}y)$ be a constructed cap automaton for the left side. $\mathcal{P}_l$ is clearly strictly epsilon free, as it has a unique initial state (x, y) and no ε-transitions. To see that it is gap universal, suppose that there is a transition from a non-final to a final state in $\mathcal{P}_l$. The second form of the **descend** rule is the only

rule which may license such a transition. It thus has the form $\mathcal{P}_l \vdash (s,u) \xrightarrow{i} (\_,v)$ for some $u,v \in V(\varphi)$, and $s \in V(\varphi) \cup \{\_\}$. The only rule which can turn $(\_,v)$ into a final state is the **top** rule, but this rule turns $(\_,v)$ directly into a universal state. (The **final states** rule does not apply because of the underscore on the left.)

To prove that $\mathcal{P}_l$ is shuffled, we assume a path π and two different states q and q' with $\mathcal{P}_l \vdash q \xleftarrow{\pi} q_0 \xrightarrow{\pi} q' \dashrightarrow q$. We unify the states q_0, q, q' with the rules of Table 7.1 and get $\mathcal{P}_l \vdash (v,u) \xleftarrow{\pi} (x,y) \xrightarrow{\pi} (u,s) \dashrightarrow (v,u)$ for some $u,v \in V(\varphi) \cup \{x,y\}$ and $s \in V(\varphi) \cup \{x,y,\_\}$. By construction of the automaton (Table 7.1), $\mathcal{P}_l \vdash (x,y) \xrightarrow{\pi} (u,u)$ must also hold. By **reflexivity** and **all**, the language $\{\pi' \mid \pi\pi' \in \mathcal{L}(\mathcal{P}_l)\}$ is universal.

We finally prove the strict cap property. All P-edges of $\mathcal{P}_l$ are of the form $\mathcal{P} \vdash (u,s) \dashrightarrow q$ for some state q. The last transition in all transition sequences reaching (u,s) must be licensed by the **descend** rule, and thus is of the form $\mathcal{P} \vdash (v,s_0) \xrightarrow{i} (u,s)$. Now, the **final states** rule applies to (v,s_0). Repeating this argument inductively shows that all states leading to (v,s_0) are final too.

8 Restricted Cap Set Expressions

We now formulate corresponding restrictions for cap set expressions. Thereby, we obtain the restrictions needed for Theorem 5.1 to hold.

Definition 8.1
We call a cap set expression over alphabet A restricted *if it is a shuffled expression with the following abstract syntax where R_1, R_2, R range over regular expressions over A:*

$$F \quad ::= \quad pr(R_1 R_2^\circ) \mid \; RA^* \mid \; F_1 \cup F_2$$

A cap set expression of sort F is called shuffled *if all its components of the form $pr(R_1 R_2^\circ)$ with $\mathcal{L}(R_2) \neq \{\varepsilon\}$ satisfy:*

shuffle: *for all words $\pi \in \mathcal{L}(R_1) \cap \mathcal{L}(R_1 R_2)$ it holds that $\pi A^* \subseteq \mathcal{L}(R_1)$.*

Proposition 8.1
Universality of restricted cap set expressions and restricted cap automata are equivalent modulo deterministic polynomial time transformations.

PROOF. It is easy to see that those cap automata are gap-universal, strictly cap, and shuffled that the proof of Proposition 6.2 constructs for restricted cap expressions. They can be made strictly ε-free in addition, as we will see in Lemma 8.2.

Conversely, given a restricted cap automaton $\mathcal{P}$, we can express the regular part of $\mathcal{P}$ by a restricted cap set expression $pr(R_1 \varepsilon^\circ) \cup R_2 A^*$, because of $\mathcal{P}$ is gap universal. The cap automaton $\mathcal{P}$ is also strictly cap, so we can translate every P-edge of $\mathcal{P}$ in a restricted cap set expression $pr(R_1 R_2^\circ)$. All build restricted cap set expressions are shuffled since $\mathcal{P}$ is shuffled.

In order to complete the preceding proof, we must show how to make cap automata strictly ε-free. We call a state q of a cap automaton $\mathcal{P}$ *normalized* if q has no in-going transitions and no out-going P-edges.

Lemma 8.1
If a cap automaton $\mathcal{P}$ has a unique initial state then this state can be normalized, while preserving the language of the automaton, gap-universality, strict cap, and the shuffle property.

PROOF. Let $\mathcal{P}$ be a cap automaton with one initial state q_0. We construct a new automaton $\mathcal{P}'$ by adding a state q_0' to $\mathcal{P}$ which inherits all out-going Δ-transitions and in-going P-edges from q_0. We let q_0' be the unique initial state of $\mathcal{P}'$. This state is normalized.

Lemma 8.2 (Epsilon elimination)
Every cap automaton can be made strictly ε-free in polynomial time, while preserving the language and the properties: gap-universal, strictly cap, and shuffle.

PROOF. First, we eliminate ε-edges in the underlying finite automaton. This yields a cap automaton which may have more than one initial state. We assume w.l.o.g that this automaton consists of n independent parts where each part has exactly one initial state. Second, we normalize all n initial states according to Lemma 8.1.

We prove that the ε-elimination does not affect the language. Let $\mathcal{P}_{\not\varepsilon}$ be the cap automaton that results after ε-elimination in a cap automaton $\mathcal{P}_\varepsilon$. By induction it holds that $\mathcal{P}_\varepsilon \vdash q_1 \xrightarrow{\pi} q_2$ if and only if $\mathcal{P}_{\not\varepsilon} \vdash q_1 \xrightarrow{\pi} q_2$ where $\pi \neq \varepsilon$. Further it holds $\mathcal{P}_\varepsilon \vdash {>}q_0 \xrightarrow{\varepsilon} q_0'$ if and only if $\mathcal{P}_{\not\varepsilon} \vdash {>}q_0'$. This implies

$$\mathcal{P}_\varepsilon \vdash {>}q_0 \xrightarrow{\pi} \underline{\underline{q_1}} \xrightarrow{\mu} q_2 \dashrightarrow \underline{\underline{q_1}} \quad \text{if and only if} \quad \mathcal{P}_{\not\varepsilon} \vdash {>}q_0' \xrightarrow{\pi} \underline{\underline{q_1}} \xrightarrow{\mu} q_2 \dashrightarrow \underline{\underline{q_1}}.$$

So $\mathcal{L}(\mathcal{P}_\varepsilon)$ and $\mathcal{L}(\mathcal{P}_{\not\varepsilon})$ are equal.

Second, we copy the whole cap automaton n-times where n is the number of the initial states. The result is a big cap automaton which consists of n independent parts, each has a single initial state.

Third, we unify all n initial states into a single initial state. The unified initial state inherits all P- and Δ-edges of the unified initial states. It is final if and only if one of the previous initial states was. Since all initial states are normalized this step does neither change the language of $\mathcal{P}$, nor gap-universal, the strict cap nor the shuffle property.

9 Back Translation for Restricted Cap Automata

We now encode universality of restricted cap automata over alphabet $\{1, \ldots, n\}$ back to NSSE over the signature $\{\bot, f, \top\}$ where $\mathit{arity}(f) = n$. Again, our construction applies to finite, regular, and possibly infinite trees.

Definition 9.1
Given a restricted cap automaton we assume two fresh variables $l(q)$ and $r(q)$ for each state $\mathcal{P} \vdash q$. The judgment $J(\mathcal{P})$ of a restricted cap automaton $\mathcal{P}$ with initial state $\mathcal{P} \vdash \ {>}q_0$ is $\varphi_\mathcal{P} \models^? l(q_0){\leq}r(q_0)$ where $\varphi_\mathcal{P}$ is the least constraint with the properties in Table 9.1.

The judgment $J(\mathcal{P})$ is defined such that $\mathcal{P}$ recognizes exactly the set of l-safe words for $J(\mathcal{P})$ whereas the set of r-safe words for $J(\mathcal{P})$ is A^*.

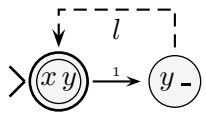
Proposition 9.1 (Correctness)
Every complete and restricted cap automaton $\mathcal{P}$ with initial state $\mathcal{P} \vdash \ {>}q_0$ over alphabet A satisfies:

$$\mathcal{L}(\mathcal{P}) \ = \ \mathcal{L}(\mathcal{P}_l(J(\mathcal{P})) \quad \textit{and} \quad A^* \ = \ \mathcal{L}(\mathcal{P}_r(J(\mathcal{P})).$$

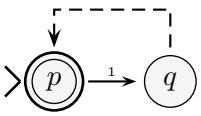
For proof we need an auxiliary lemma.

left	$l(q){\leq}f(l(q_1), \ldots, l(q_n))$ in $\varphi_\mathcal{P}$	if $\mathcal{P} \vdash \underline{q} \xrightarrow{i} q_i$ for all $1 \leq i \leq n$.
right	$f(r(q_1), \ldots, r(q_n)){\leq}r(q)$ in $\varphi_\mathcal{P}$	if $\mathcal{P} \vdash q \xrightarrow{i} q_i$ for all $1 \leq i \leq n$
top	$r(q'){=}\top$ in $\varphi_\mathcal{P}$	if $\mathcal{P} \vdash q \xrightarrow{i} \underline{\underline{q'}}$, q not final
P-edges	$l(q){\leq}i[\, r(q_2)\,]$ in $\varphi_\mathcal{P}$	if $\mathcal{P} \vdash \underline{q} \xrightarrow{i} q_1 \dashrightarrow q_2,\ q_1{\neq}q_2$

Table 9.1: Back translation: the constraint $\varphi_\mathcal{P}$ of a restricted cap automaton $\mathcal{P}$.



$x \leq f(y) \models^? x \leq y$



$l(p) \leq f(l(q)) \wedge f(r(q)) \leq r(p) \wedge l(p) \leq f(r(p)) \models^? l(p) \leq r(p)$

Figure 9.1: A judgment, its pair of cap automata, and the back translation of the left cap automaton.

Lemma 9.1
If $J(\mathcal{P}) \vdash u \leq v$ then $u = v$.

PROOF. By structural induction on derivations $J(\mathcal{P}) \vdash u \leq v$. The derivation rules are given in Table 3.1. The consideration for the rules **reflexive** and **trans.** are obvious. So, we can restrict ourself to the **decomp.** rule. So assume that the decomposition rule derives $J(\mathcal{P}) \vdash u \leq v$. Hence, $f(\ldots) \leq u$ and $v \leq f(\ldots)$ in $J(\mathcal{P})$ while $J(\mathcal{P}) \vdash u \leq v$. The induction hypothesis yields $u = v$. The construction of $J(\mathcal{P})$ in Table 9.1 implies that $u = r(q)$ and $v = l(q)$ for some states p, q. Hence $r(q) = l(q)$ which is impossible so that the **decomp.** rule cannot be applicable.

PROOF(Proof of Proposition 9.1).

1. The language $\mathcal{L}(\mathcal{P}_r(J(\mathcal{P})))$ is universal: Since $\mathcal{P}$ is complete such that the **right** rule implies for all words $\pi \in A^*$ that there exists a state $\mathcal{P} \vdash q$ satisfying $\varphi_{\mathcal{P}} \models \pi(r(q)) \leq r(q_0)$. Thus, $\mathcal{P}_r(J(\mathcal{P})) \vdash \underline{\underline{(l(q_0), r(q_0))}} \xrightarrow{\pi} \underline{\underline{(\_, l(q))}}$ by the second case of the **descend** rule, i.e. π is accepted by $\mathcal{P}_r(J(\mathcal{P}))$.

2. We omit the proof for $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P}_l(J(\mathcal{P})))$ which only requires the completeness of $\mathcal{P}$ and the strictly cap property.

3. The remaining inclusion $\mathcal{L}(\mathcal{P}_l(J(\mathcal{P}))) \subseteq \mathcal{L}(\mathcal{P})$ is most interesting. Note that according to Lemma 9.1 we have not to consider any support $J(\mathcal{P}) \vdash u \leq v$ in the construction of the appendant cap automata $\mathcal{P}_l(J(\mathcal{P}))$. We start with an auxiliary **claim**: If $\mathcal{P}$ provides transitions

$$\mathcal{P}_l(J(\mathcal{P}) \vdash (l(q_0), s_0) \xrightarrow{\pi} (l(q_n), s_n) \xrightarrow{i} (l(q), s)$$

then there exist transitions $\mathcal{P} \vdash \underline{\underline{q_0}} \xrightarrow{\pi} \underline{\underline{q_n}}$. This claim can be proved as follows: All transitions must be licensed by a constraint in $\varphi_{\mathcal{P}}$ which is of the form $l(q_i) \leq f(\ldots, l(q_{i+1}), \ldots)$ where $1 \leq i \leq n$. Such constraints can only be created by the **left** rule. There thus exist transitions $\mathcal{P} \vdash \underline{\underline{q_0}} \xrightarrow{\pi} q_n$ such $\mathcal{P} \vdash \underline{\underline{q_i}}$ for all $0 \leq i < n$. We can infer $\mathcal{P} \vdash \underline{\underline{q_n}}$ as required.

We now come back to the main proof. Suppose $\pi \in \mathcal{L}(\mathcal{P}_l(J(\mathcal{P})))$. There are three kinds of transitions by which π can be recognized.

a) We first consider transitions using the **reflexivity** rule to recognize π. These contain a transition sequence of the following form for some prefix $\pi' \leq \pi$:

$$\mathcal{P}_l(J(\mathcal{P})) \vdash (l(q_0), r(q_0)) \xrightarrow{\pi'} \underline{\underline{(r(q_n), r(q_n))}}$$

Either $\pi' = \pi$ or this sequence can be continued to recognize π in the state $\underline{\underline{all}}$. The first continuation step is by the **reflexivity** rule itself and all subsequent steps are due to the **all** rule.

Note that $n \geq 1$. We first consider the descendants on the left hand side, which starts from state $l(r_0)$ and continues over $l(l_{n-1})$ to $r(q_m)$. The last step must be induced by a constraint in $\varphi_{\mathcal{P}}$ that is contributed by the **P-edges** rule. This and the preceding claim yield the existence of the following transitions for some state $q \neq q_n$:

$$\mathcal{P} \vdash q_0 \xrightarrow{\pi'} q \dashrightarrow q_n$$

We next consider the descendants on the right hand side. They must be induced by constraints in $\varphi_{\mathcal{P}}$ that are inherited form the following transition sequence:

$$\mathcal{P} \vdash q_0 \xrightarrow{\pi'} q_n$$

Now we can apply that $\mathcal{P}$ is shuffled which shows that the language $\{\pi'' \mid \pi'\pi'' \in \mathcal{L}(\mathcal{P})\}$ is universal (since $q \neq q_n$)). Thus, $\pi \in \mathcal{L}(\mathcal{P})$ as required.

b) Second, we consider transitions using the **top** rule. These contain a part of the following form for some prefix $\pi' \leq \pi$ and such that $r(q_n){=}\top$ in $\varphi_{\mathcal{P}}$.

$$\mathcal{P}_l(J(\mathcal{P})) \vdash (l(q_0), r(q_0)) \xrightarrow{\pi'} \underline{\underline{(s_n, r(q_n))}}$$

Again, either $\pi' = \pi$ or this sequence can be continued to recognize π in the state $\underline{\underline{all}}$. The first continuation step is by the **top** rule itself and all subsequent steps are due to the **all** rule.

The above transitions of $\mathcal{P}_l(J(\mathcal{P}))$ are induced by the following transition sequence in $\mathcal{P}$ where q_{n-1} is not final:

$$\mathcal{P} \vdash q_0 \xrightarrow{\pi'} \underline{\underline{q_n}}$$

The gap universal property which holds for $\mathcal{P}$ by assumption yields that $\{\pi'' \mid \mathcal{P} \vdash q_n \xrightarrow{\pi''} \underline{\underline{q_n}}\}$ is universal. Thus, $\pi \in \mathcal{L}(\mathcal{P})$.

c) Third, we consider the last case where the class of π is A in $\mathcal{P}_l(J(\mathcal{P}))$. The recognizing transition has to apply the rule for **final states**:

$$\mathcal{P}_l(J(\mathcal{P})) \vdash \underline{\underline{(l(q_0), r(q_0))}} \xrightarrow{\pi} \underline{\underline{(l(q_n), s_n)}} \xrightarrow{i} (\theta(q), p(\pi))$$

All transitions except the last one must be contributed by the **left** rule. The **P-edges** can only apply at the end. In this case however, we can freely exchange the last transition by another using the **left** rule as well. Given this, we can apply our initial claim which yields:

$$\mathcal{P} \vdash \underline{\underline{q_0}} \xrightarrow{\pi} \underline{\underline{q_n}}$$

Thus, we have shown that $\pi \in \mathcal{L}(\mathcal{P})$ for this case too.

d) Finally, we have to consider transitions that recognize π through P-edges of $\mathcal{P}_l(J(\mathcal{P}))$. Here we have transitions where π is a prefix of $\pi_1\pi_2^k$ for some $k \geq 0$: $\mathcal{P}_l(J(\mathcal{P})) \vdash$

$$(l(q_0), r(q_0)) \xrightarrow{\pi_1} (l(q_i), r(q_i)) \xrightarrow{\pi_2} (r(q_n), s_n) \dashrightarrow (l(q_i), r(q_i))$$

The **P-edges** rule in the construction of $\mathcal{P}_l$ requires $q_n = q_i$. The automaton $\mathcal{P}$ thus has the following transitions for some state q:

$$\mathcal{P} \vdash q_0 \xrightarrow{\pi_1} q_i \xrightarrow{\pi_2} q \dashrightarrow q_i$$

This transition and the strictly cap property allows $\mathcal{P}$ to recognize all prefixes of $\pi_1\pi_2^k$ for all $k \geq 0$, i.e. $\pi \in \mathcal{L}(\mathcal{P})$.

For illustration, we reconstruct an entailment judgment for $\mathcal{P}_l(x{\leq}f(y) \models^? x{\leq}y)$ given in Table 9.1. Before we start we rename the states of $\mathcal{P}_l(x{\leq}f(y) \models^? x{\leq}y)$ to p and q. We translate the edge $\underline{\underline{p}} \xrightarrow{1} q$ to the constraint $l(p){\leq}f(l(q)) \wedge f(r(q)){\leq}r(p)$ (rule **left** and **right** of Table 9.1). The rule **P-edges** maps the P-edge $q \dashrightarrow p$ to the constraint $l(p){\leq}f(r(p))$. If we now construct the left automaton of the computed constraint, we get the original automaton back.

Lemma 9.2
Let $\mathcal{P}$ be a restricted cap automaton with initial state $\mathcal{P} \vdash \succ q$. The constructed constraint $\varphi_{\mathcal{P}}$ is satisfiable over finite and infinite trees.

THEOREM 9.1 (Back translation)
Universality of restricted cap automata over the alphabet $\{1, \ldots, arity(f)\}$ can be reduced in polynomial time to NSSE with signature $\{\bot, f, \top\}$ (respectively over finite, regular, or possibly infinite trees).

PROOF. Let $\mathcal{P}$ be complete and restricted cap automaton. Universality of $\mathcal{L}(\mathcal{P})$ is equivalent to universality of both languages: $\mathcal{L}(\mathcal{P}_l(J(\mathcal{P})))$ and $\mathcal{L}(\mathcal{P}_r(J(\mathcal{P})))$ (Proposition 9.1). Since $\varphi_{\mathcal{P}}$ is clash-free (Lemma 9.2), the latter is equivalent to that NSSE holds for the judgment $J(\mathcal{P})$ (Theorem 7.1).

PROOF(Proof of Proposition 9.1).

1. The language $\mathcal{L}(\mathcal{P}_r(J(\mathcal{P})))$ is universal: Since $\mathcal{P}$ is complete such that the **right** rule implies for all words $\pi \in A^*$ that there exists a state $\mathcal{P} \vdash q$ satisfying $\varphi_{\mathcal{P}} \models \pi(r(q)) \leq r(q_0)$. Thus, $\mathcal{P}_r(J(\mathcal{P})) \vdash \underline{\underline{(l(q_0), r(q_0))}} \xrightarrow{\pi} \underline{\underline{(\_, l(q))}}$ by the second case of the **descend** rule, i.e. π is accepted by $\mathcal{P}_r(J(\mathcal{P}))$.

2. We omit the proof for $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P}_l(J(\mathcal{P})))$ which only requires the completeness of $\mathcal{P}$ and the strictly cap property.

3. The remaining inclusion $\mathcal{L}(\mathcal{P}_l(J(\mathcal{P}))) \subseteq \mathcal{L}(\mathcal{P})$ is most interesting. Note that according to Lemma 9.1 we have not to consider any support $J(\mathcal{P}) \vdash u \leq v$ in the construction of the appendant cap automata $\mathcal{P}_l(J(\mathcal{P}))$. We start with an auxiliary **claim**: If $\mathcal{P}$ provides transitions

$$\mathcal{P}_l(J(\mathcal{P}) \vdash (l(q_0), s_0) \xrightarrow{\pi} (l(q_n), s_n) \xrightarrow{i} (l(q), s)$$

then there exist transitions $\mathcal{P} \vdash \underline{\underline{q_0}} \xrightarrow{\pi} \underline{\underline{q_n}}$. This claim can be proved as follows: All transitions must be licensed by a constraint in $\varphi_{\mathcal{P}}$ which is of the form $l(q_i) \leq f(\ldots, l(q_{i+1}), \ldots)$ where $1 \leq i \leq n$. Such constraints can only be created by the **left** rule. There thus exist transitions $\mathcal{P} \vdash \underline{\underline{q_0}} \xrightarrow{\pi} q_n$ such $\mathcal{P} \vdash \underline{\underline{q_i}}$ for all $0 \leq i < n$. We can infer $\mathcal{P} \vdash \underline{\underline{q_n}}$ as required.

We now come back to the main proof. Suppose $\pi \in \mathcal{L}(\mathcal{P}_l(J(\mathcal{P})))$. There are three kinds of transitions by which π can be recognized.

a) We first consider transitions using the **reflexivity** rule to recognize π. These contain a transition sequence of the following form for some prefix $\pi' \leq \pi$:

$$\mathcal{P}_l(J(\mathcal{P})) \vdash (l(q_0), r(q_0)) \xrightarrow{\pi'} \underline{\underline{(r(q_n), r(q_n))}}$$

Either $\pi' = \pi$ or this sequence can be continued to recognize π in the state $\underline{\underline{all}}$. The first continuation step is by the **reflexivity** rule itself and all subsequent steps are due to the **all** rule.

Note that $n \geq 1$. We first consider the descendants on the left hand side, which starts from state $l(r_0)$ and continues over $l(l_{n-1})$ to $r(q_m)$. The last step must be induced by a constraint in $\varphi_{\mathcal{P}}$ that is contributed by

the **P-edges** rule. This and the preceding claim yield the existence of the following transitions for some state $q \neq q_n$:

$$\mathcal{P} \vdash q_0 \xrightarrow{\pi'} q \dashrightarrow q_n$$

We next consider the descendants on the right hand side. They must be induced by constraints in $\varphi_{\mathcal{P}}$ that are inherited form the following transition sequence:

$$\mathcal{P} \vdash q_0 \xrightarrow{\pi'} q_n$$

Now we can apply that $\mathcal{P}$ is shuffled which shows that the language $\{\pi'' \mid \pi'\pi'' \in \mathcal{L}(\mathcal{P})\}$ is universal (since $q \neq q_n$)). Thus, $\pi \in \mathcal{L}(\mathcal{P})$ as required.

b) Second, we consider transitions using the **top** rule. These contain a part of the following form for some prefix $\pi' \leq \pi$ and such that $r(q_n)=\top$ in $\varphi_{\mathcal{P}}$.

$$\mathcal{P}_l(J(\mathcal{P})) \vdash (l(q_0), r(q_0)) \xrightarrow{\pi'} \underline{\underline{(s_n, r(q_n))}}$$

Again, either $\pi' = \pi$ or this sequence can be continued to recognize π in the state $\underline{\underline{all}}$. The first continuation step is by the **top** rule itself and all subsequent steps are due to the **all** rule.

The above transitions of $\mathcal{P}_l(J(\mathcal{P}))$ are induced by the following transition sequence in $\mathcal{P}$ where q_{n-1} is not final:

$$\mathcal{P} \vdash q_0 \xrightarrow{\pi'} \underline{\underline{q_n}}$$

The gap universal property which holds for $\mathcal{P}$ by assumption yields that $\{\pi'' \mid \mathcal{P} \vdash q_n \xrightarrow{\pi''} \underline{\underline{q_n}}\}$ is universal. Thus, $\pi \in \mathcal{L}(\mathcal{P})$.

c) Third, we consider the last case where the class of π is A in $\mathcal{P}_l(J(\mathcal{P}))$. The recognizing transition has to apply the rule for **final states**:

$$\mathcal{P}_l(J(\mathcal{P})) \vdash \underline{\underline{(l(q_0), r(q_0))}} \xrightarrow{\pi} \underline{\underline{(l(q_n), s_n))}} \xrightarrow{i} (\theta(q), p(\pi))$$

All transitions except the last one must be contributed by the **left** rule. The **P-edges** can only apply at the end. In this case however, we can freely exchange the last transition by another using the **left** rule as well. Given this, we can apply our initial claim which yields:

$$\mathcal{P} \vdash \underline{\underline{q_0}} \xrightarrow{\pi} \underline{\underline{q_n}}$$

Thus, we have shown that $\pi \in \mathcal{L}(\mathcal{P})$ for this case too.

d) Finally, we have to consider transitions that recognize π through P-edges of $\mathcal{P}_l(J(\mathcal{P}))$. Here we have transitions where π is a prefix of $\pi_1\pi_2^k$ for some $k \geq 0$: $\mathcal{P}_l(J(\mathcal{P})) \vdash$

$$(l(q_0), r(q_0)) \xrightarrow{\pi_1} (l(q_i), r(q_i)) \xrightarrow{\pi_2} (r(q_n), s_n) \dashrightarrow (l(q_i), r(q_i))$$

The **P-edges** rule in the construction of $\mathcal{P}_l$ requires $q_n = q_i$. The automaton $\mathcal{P}$ thus has the following transitions for some state q:

$$\mathcal{P} \vdash q_0 \xrightarrow{\pi_1} q_i \xrightarrow{\pi_2} q \dashrightarrow q_i$$

This transition and the strictly cap property allows $\mathcal{P}$ to recognize all prefixes of $\pi_1\pi_2^k$ for all $k \geq 0$, i.e. $\pi \in \mathcal{L}(\mathcal{P})$.

10 Equivalence of Variants

10.1 Arity equivalence

We prove Corollary 5.1 which states that all variants of NSSE over the signature $\{\bot, f, \top\}$ are equivalent if the arity of f is at least 2. Given the characterization of NSSE in Theorem 5.1 it remains to prove a corresponding result for restricted cap automata:

Proposition 10.1
The universality problems of restricted cap automata over the alphabet $\{1, \ldots, n\}$ are equivalent for all $n \geq 2$ modulo polynomial time transformations.

PROOF. We first show how to extend to alphabet. Consider a restricted cap automaton $\mathcal{P}$ and an alphabet $A = \{1, \ldots, n-1\}$. We construct another restricted cap automaton $\mathcal{P}'$ with an alphabet $A' = \{1, \ldots, n\}$ in linear time. The cap automaton $\mathcal{P}'$ is identical to $\mathcal{P}$ up to the additions

$$\mathcal{P}' \vdash q_0 \xrightarrow{n} \underline{\underline{q_2}} \xrightarrow{1,\ldots,n} \underline{\underline{q_2}} \qquad \mathcal{P}' \vdash q_0 \xrightarrow{1,\ldots,n} q_1 \xrightarrow{1,\ldots,n} q_1 \xrightarrow{n} \underline{\underline{q_2}} \xrightarrow{1,\ldots,n} \underline{\underline{q_2}}$$

where q_0 is the initial state of $\mathcal{P}$ and $\mathcal{P}'$ and q_1, q_2 are two fresh states. This construction composes:

$$\mathcal{L}(\mathcal{P}') = \{\ \pi\sigma \mid \pi \in \mathcal{L}(\mathcal{P}), \quad \text{and } \sigma \in n(1, \ldots, n)^*\ \}.$$

We now consider alphabet restriction. Let $\mathcal{P}$ be a restricted cap automaton with alphabet $A = \{1, \ldots, n^2\}$ where $n^2 \geq 3$. We can assume w.l.o.g that A is of that form. Otherwise we can increase A by the previous construction until this form is reached.

We next construct a restricted cap automaton $\mathcal{P}'$ with alphabet $A' = \{1, \ldots, n\}$ in polynomial time such that $\mathcal{L}(\mathcal{P})$ is universal if and only if $\mathcal{L}(\mathcal{P}')$ is universal. We encode a letter of A in two letters of A' to the base n via the standard encoding $d : A \to A' \times A'$:

$$d(i) = (\, d_1(i)\,,\, d_2(i)\,) = \left(\, \left\lfloor \frac{i}{n} \right\rfloor \,,\, i \bmod n \right).$$

The cap automaton $\mathcal{P}'$ has two states q and q' for every state q of $\mathcal{P}$. The states q and q' are final in $\mathcal{P}'$ if q is final in $\mathcal{P}$, i.e.

$$\mathcal{P}' \vdash \underline{\underline{q_1}} \text{ and } \mathcal{P}' \vdash \underline{\underline{q_1'}} \quad \text{if} \quad \mathcal{P} \vdash \underline{\underline{q_1}}.$$

The cap automaton $\mathcal{P}$ and $\mathcal{P}'$ share the same initial state and the same P-edges. We define the transitions of $\mathcal{P}'$ by

$$\mathcal{P}' \vdash q_1 \xrightarrow{d_1(i)} q_1' \xrightarrow{d_2(i)} q_2 \quad \text{if} \quad \mathcal{P} \vdash q_1 \xrightarrow{i} q_2.$$

We can show by induction that the word $i_1 \dots i_m$ is in $\mathcal{L}(\mathcal{P})$ if and only if the words $d_1(1)\, d_2(1) \dots d_1(m-1)\, d_2(m-1)\, d_1(m)$ and $d_1(1)\, d_2(1) \dots d_1(m)\, d_2(m)$ are in $\mathcal{L}(\mathcal{P}')$.

11 A Decidable Fragment of NSSE

We prove a subclass of non-structural subtype entailment (NSSE) to be decidable by reducing it to satisfiability of word equations with regular constraints. The latter has been shown to be decidable [Sch91] and even PSPACE*-complete [Pla99]. The embedding is based on our entailment characterization by cap set expressions. This new technique is unrelated to the methods used in [NP99] where we have proved another fragment of entailment to be decidable.*

We have proved NSSE to be equivalent to the universality of restricted cap set expressions (Theorem 5.1). In this chapter we solve universality for cap set expressions

$$\bigcup_i R_i S_i^\circ$$

build over regular expressions R_i, S_i which are restricted to a new property that we shall now call straight. This new fragment is rather unrelated to restricted cap set expressions given in Section 8.

A regular expression R is called straight if

straight: there are no words π, μ with $\mu \neq \varepsilon$ and $\pi\mu^* \subseteq \mathcal{L}(R)$.

Straight regular expressions are exactly those which hold a representation of a deterministic finite automaton where all finite nodes are cycle free, what means there is no transition sequence from one final node to itself. For example 1^*2 is straight and 1^* is not.

THEOREM 11.1
Universality of cap set expressions over straight regular expressions is decidable.

For its proof we reduce the stated problem to satisfiability of word equations with regular constraints.

We define word equations as an expression of the form $\pi_1 = \pi_2$, where π_i are non-empty words in a mixed alphabet of constants in a finite signature Σ and an enumerable set of variables. We denote constants by natural numbers $1, 2, \ldots$ and variables by x, y, z, u, v, w. A solution h of a word equation $\pi_1 = \pi_2$ is an assignment of values in

conjunction	$x{=}y \wedge u{=}v$	$\leftrightarrow$	$x1ux2u{=}y1vy2v$ for $1,2 \in \Sigma$
disjunction	$x{=}y \vee u{=}v$	$\leftrightarrow$	$(x{\leq}y \vee u \leq v) \wedge (x{\leq}y \vee v{\leq}u) \wedge$ $(y{\leq}x \vee u \leq v) \wedge (y{\leq}x \vee v{\leq}u)$
prefix of	$x{\leq}y$	$\leftrightarrow$	$\exists z.\ xz{=}y$
prefix disjunction	$x{\leq}y \vee u{\leq}v$	$\leftrightarrow$	$\exists z, x', y' x{=}zx' \wedge y{=}zy' \wedge$ $\exists w, u', v' u{=}wu' \wedge v{=}wv' \wedge (x'{=}\varepsilon \vee u'{=}\varepsilon)$
epsilon disjunction	$x{=}\varepsilon \vee y{=}\varepsilon$	$\leftrightarrow$	$xy{=}yx \wedge \exists u, u', v, v'.\ uu'{=}1 \wedge vv'{=}2$ $\wedge xuyu'{=}yu'xu \wedge xvyv'{=}yv'xv$ for $1,2 \in \Sigma$
negation	$x{\neq}y$	$\leftrightarrow$	$x{<}y \vee y{<}x \vee x{\not\leq}y$
proper prefix of	$x{<}y$	$\leftrightarrow$	$\bigvee_{i\in\Sigma} \exists xi{\leq}y$
incomparable	$x{\not\leq}y$	$\leftrightarrow$	$\bigvee_{i,j\in\Sigma: i\neq j} \exists z.\ zi{\leq}x \wedge zj{\leq}y$

Table 11.1: Word equations are closed under Boolean compositions.

Σ^* to variables such that π_1 and π_2 become syntactically identical if all its variables are replaced by the corresponding values. Variable assignments can be restricted to hold only values given in supplementary regular constraints.

We say that a relation $R \subseteq (\Sigma^*)^n$ is expressible by a word equation $\pi_1{=}\pi_2$ over variables $x_1, \ldots x_{n+m}$ if

$$R \;=\; \{\, (\, h(x_1), \ldots, h(x_n)\,) \mid h \text{ is a solution of } \exists x_{n+1}, \ldots, x_{n+m} : \pi_1{=}\pi_2 \,\}.$$

For instance the prefix relation $x \in pr(y)$ could be expressed by a word equation $xz{=}y$ where z is existential quantified. Alternatively we denote this word equation by $x{\leq}y$ where we have not to explicitly name a fresh variable z. For an overview which other relations are expressible by word equations and which are not we refer to [KMP00].

It is well known that word equations are closed under Boolean compositions [BS86]. Table 11.1 recapitulates all encodings: disjunction of word equation is reduced to elementary disjunctions over tests $x{\neq}\varepsilon$ which could be further reduced to conjunctions of word equations [Ang79]; negation is reduced to the incomparable relation $x{\not\leq}y$ which says that $x{\not\leq}y$ and $y{\not\leq}x$.

Universality does not hold if and only if there exists a path π which does not belong to the cap set language:

$$\exists \pi. \bigwedge_i \pi \notin R_i S_i^{\circ}.$$

Since word equations are closed under conjunctions (Table 11.1) it remains to express subformulas $\pi \notin RS^{\circ}$ by word equations with regular constraints. Note that $\pi \notin RS^{\circ}$

empty set	$pr(\emptyset)$	$=_{\mathrm{df}}$	$\emptyset$	
epsilon	$pr(\varepsilon)$	$=_{\mathrm{df}}$	ε	
letter	$pr(a)$	$=_{\mathrm{df}}$	$a \cup \varepsilon$	where $a \in A$
union	$pr(R_1 \cup R_2)$	$=_{\mathrm{df}}$	$pr(R_1) \cup pr(R_2)$	
concatenation	$pr(R_1 R_2)$	$=_{\mathrm{df}}$	$(R_1 pr(R_2)) \cup pr(R_1)$	
Kleene star	$pr(R^*)$	$=_{\mathrm{df}}$	$R^* pr(R)$	

Table 11.2: Regular expressions are closed under prefix closure.

holds if and only if for all $\mu \in R$, $\nu \in S$ it holds that $\pi \notin \mu\nu^\circ$. These two all-quantifiers inhibit a translation for full cap set expressions. We work around by the following property:

Lemma 11.1 (Uniqueness of decomposing straight regular sets)
Let π_1, μ_1 be words of a straight regular set R_1 and let π_2, μ_2 be words of an arbitrary regular set R_2. Then, $\pi_1\pi_2 = \mu_1\mu_2$ implies $\pi_1 = \mu_1$ and $\pi_2 = \mu_2$.

PROOF. We consider two deterministic finite automata representations $\mathcal{A}_{R_i}$ for R_i, respectively. Since R_1 is straight we assume that all final nodes in $\mathcal{A}_{R_1}$ are cycle free. We can build a finite automaton $\mathcal{A}_{R_1}\mathcal{A}_{R_2}$ accepting the language R_1R_2 which is still deterministic. Hereunto we merge all final nodes of $\mathcal{A}_{R_1}$ and the one start node of $\mathcal{A}_{R_2}$ to one new node inheriting all their edges. Since $\mathcal{A}_{R_1}\mathcal{A}_{R_2}$ is deterministic there is only one unique way to accept a word $\pi_1\pi_2$ where $\pi_i \in \mathcal{A}_{R_i}$.

This essential property of straight regular expressions allows us to express $\pi \notin RS^\circ$ by existential quantified word equations with regular constraints.

Lemma 11.2
Let R, S be straight regular expressions , then $\pi \notin RS^\circ$ if and only if

$$\pi \notin R\,pr(S) \text{ and } (\, pr(\pi) \cap RS = \emptyset \text{ or } \exists \pi_1 \in R,\, \pi_2 \in S,\, \pi_3 \notin \pi_2^\circ : \pi = \pi_1\pi_2\pi_3 \,).$$

PROOF. Since $R\,pr(S) \subseteq RS^\circ$ we are able to split our test $\pi \notin RS^\circ$ into three parts: (1) $\pi \notin R\,pr(S)$ and ((2) $pr(\pi) \cap RS \neq \emptyset$ implies (3) $\pi \notin RS^\circ$). We have already seen that (3) is equivalent to the assertion that for all $\pi_1 \in R$, $\pi_2 \in S$: $\pi \notin \pi_1\pi_2^\circ$. Under the assumption of (2), we rewrite (3) as

$$\text{for all and at least one } \pi_1 \in R,\, \pi_2 \in S,\, \pi_3 : \pi = \pi_1\pi_2\pi_3 \wedge \pi_3 \notin \pi_2^\circ.$$

to be in cap	$x \in y^\circ$ for $y \neq \varepsilon$	$\leftrightarrow$	$x \leq yx$
not to be in cap	$x \notin y^\circ$ for $y \neq \varepsilon$	$\leftrightarrow$	$\exists x'.\ x' \in y^\circ \wedge x \not\leqslant x'$

Table 11.3: Word equations express positive and negative cap set membership.

Since R and S are both straight we can apply Lemma 11.1 which says that the segmentations of π in $\pi = \pi_1\pi_2\pi_3$ is unique and "for all and at least one" quantifiers can be safely reduced to existential quantifiers.

It remains to show that the characterization of $\pi \notin RS^\circ$ given in the previous Lemma 11.2 can be translated to word equations. For this we assume $\varepsilon \notin S$, otherwise we decompose the problem into two parts

$$\pi \notin R \ \wedge \ \pi \notin R(S - \{\varepsilon\})^\circ$$

and express $\pi \notin R$ by a regular constraint $\pi \in \bar{R}$ where $\bar{R}$ is the complement regular expression of R. The computation of $\bar{R}$ requires exponential space which yields an *EXPSPACE* algorithm assuming satisfiability of word equations with regular constraints is in *PSPACE* [Pla99].

A test of the form $\pi \notin R\,pr(S)$ can be translated just as well since the prefix closure pr(S) of a regular expression can be expressed by a further regular expression (Table 11.2). We express the condition $pr(\pi) \cap RS = \emptyset$ by a regular constraint $\pi \notin RSS'$ where S' is the regular expression containing all words over Σ. Finally we express $\pi_3 \notin (\pi_2)^\circ$ by a positive test $\pi_3 \in (\pi_2)^\circ$ (see Table 11.3) which is equivalent to $\pi_3 \leq \pi_2\pi_3$ (Lemma 5.1 in Characterization Chapter 5).

12 Conclusion and Future Work

We have given a complete characterization of the complexity of subtype satisfiability over posets through a new connection of subtype satisfiability with modal logics, which contain well understood satisfiability problems. Our technique yields a uniform and systematic treatment of different choices of subtype orderings: finite versus recursive types, structural versus non-structural subtyping, and considerations of symbols with co- and contra-variant arguments.

The complexity of modal logic satisfiability is a well known issue [Spa93]. According to this connection it applies to our origin subtype problem.

Our technique however, does not extend beyond satisfiability to other first-order fragments that require negations, such as subtype entailment, whose decidability is a longstanding open problem over non-structural signatures. Negations can certainly be modeled by our modal logic, but only over uniform signatures. In fact, there must not exist any reductions from standard signatures to uniform ones that preserve subtype entailment, for example. Otherwise, such a reduction would have implied that the first-order theory of non-structural subtyping, which is undecidable [SAN$^+$02], was a fragment of S2S, which is decidable [Rab69].

We have characterized non-structural subtype entailment (NSSE) equivalently by using regular expressions and word equations. This explains why NSSE is so difficult to solve and links NSSE to the area of string unification where powerful proof methods are available. Given that NSSE is equivalent to universality of restricted cap set expressions, one cannot expect to solve NSSE without treating word equations.

We have also shown that of all the variants of NSSE with a single function symbol of arity, at least two are equivalent modulo polynomial time transformations. One might also wish to extend the presented characterization to richer signatures. For instance, it should be possible to treat NSSE with a contra-variant function symbol. Yet the problem of how to deal with more than one non-constant function symbol is much less obvious.

Another open question is whether there exists a direct relation between cap automata and tuple tree automata with equality tests, which are used in the alternative approach to subtype entailment in [SAN$^+$02].

Finally, we have applied our NSSE characterization and proved a fragment to be decidable by reducing it to satisfiability of word equations with regular constraints. It is left open to future work whether this embedding can be extended to full NSSE.

13 Bibliography

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[AC96] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer-Verlag, 1996.

[Ang79] Dana Angluin. Finding patterns common to a set of strings. In *ACM symposium on Theory of Computing*, pages 130–141. ACM Press, 1979.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional programming languages and computer architecture*, pages 31–41. ACM Press, 1993.

[AWP99] Alexander Aiken, Edward L. Wimmers, and Jens Palsberg. Optimal representations of polymorphic types with subtyping. *Higher-Order and Symbolic Computation*, 12(3):237–282, 1999.

[BAHP82] Mordechai Ben-Ari, Joseph Y. Halpern, and Amir Pnueli. Deterministic propositional dynamic logic: finite models, complexity, and completeness. *Journal of Computer and System Sciences*, 25(3):402–417, 1982.

[BdRV01] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal logic*. Cambridge University Press, 2001.

[Ben97] Marcin Benke. *Complexity of type reconstruction in programming languages with subtyping*. PhD thesis, University Warsaw, 1997.

[Ben99] Marcin Benke. Some complexity bounds for subtype inequalities. *Theoretical Computer Science*, 212(1):3–27, 1999.

[BGM03] Patrick Blackburn, Bertrand Gaiffe, and Maarten Marx. Variable free reasoning on finite trees. In *Mathematics of Language*, pages 17–30. Indiana University, Indiana, 2003.

[BMV94] Patrick Blackburn and Wilfried Meyer-Viol. Linguistics, logic, and finite trees. *Logic Journal of the IGPL*, 2:3–29, 1994.

[BS86] J. Richard Büchi and Steven Senger. Coding in the existential theory of concatenation. *Archiv für mathematische Logik und Grundlagenforschung*, 26(7):101–106, 1986.

[BS94] Franz Baader and Jörg H. Siekmann. Unification theory. *Handbook of logic in artificial intelligence and logic programming*, 2:41–125, 1994.

[BS96] Franz Baader and Klaus Schulz. Unification in the union of disjoint equational theories: combining decision procedures. *Journal of Symbolic Computation*, 21:211–243, 1996.

[Büc60] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11, 1960.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2-3):138–164, 1988.

[CDG+02] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available at `http://www.grappa.univ-lille3.fr/tata`, 2002.

[CM94] Luca Cardelli and John C. Mitchell. Operations on records. In *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 295–350. The MIT Press, 1994.

[CMMS94] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1):4–56, 1994.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM Press, 1982.

[DN66] Ole-Johan Dahl and Kristen Nygaard. Simula: an Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[Dur95] V. Durnev. Unsolvability of positive $\forall\exists^3$-theory of free groups. In *Sibirsky mathematichesky jurnal*, volume 36(5), pages 1067–1080, 1995. In Russian, also exists in English translation.

[EST95a] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 169–184. ACM Press, 1995.

[EST95b] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to object-oriented programming. *Electronic Notes in Theoretical Computer Science*, 1, 1995.

[FA96] Manuel Fähndrich and Alexander Aiken. Making set-constraint program analyses scale. In *Workshop on Set Constraints at CP'96*. Cambridge, MA, 1996. Available as Technical Report CSD-96-917, University of California at Berkeley.

[FCB02] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[FF99] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, 1999.

[FL79] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18:194–211, 1979.

[FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, 1990.

[Fre] Alexander Frey. The Jazz programming language. Available at `www.exalead.com/jazz/`. CMA, École des Mines de Paris.

[Fre02] Alexandre Frey. Satisfying subtype inequalities in polynomial space. *Theoretical Computer Science*, 277:105–117, 2002.

[GJS96] James Gosling, Bill Joy, and Guy Steele. The Java language specification, 1996.

[GM96] Joseph Goguen and Grant Malcolm. *Algebraic semantics of imperative programs*. MIT Press, 1996.

[Hem00] Edith Hemaspaandra. The complexity of poor man's logic. In *International Symposium on Theoretical Aspects of Computer Software*, pages 230–242, 2000.

[Hin69] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic logic*. The MIT Press, 2000.

[HM95] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *POPL*, 176–185, 1995.

[HR97] Fritz Henglein and Jakob Rehof. The complexity of subtype entailment for simple types. In *IEEE Symposium on Logic in Computer Science*, pages 362–372. IEEE Computer Society Press, 1997.

[HR98] Fritz Henglein and Jakob Rehof. Constraint automata and the complexity of recursive subtype entailment. In *International Colloquium on Automata, Languages, & Programming*, Lecture Notes in Computer Science 1443, pages 616–627. Springer-Verlag, 1998.

[Jon03] Simon Peyton Jones. *Haskell 98 language and libraries*. Cambridge University Press, 2003.

[JP99] Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping. Unpublished, available at author's homepage, 1999.

[KMP00] Juhani Karhumäki, Filippo Mignosi, and Wojciech Plandowski. The expressibility of languages and relations by word equations. *Journal of the ACM*, 47(3):483–505, 2000.

[KPS94] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994.

[KPS95] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5:1–13, 1995.

[KR03] Viktor Kuncak and Martin Rinard. Structural subtyping of non-recursive types is decidable. In *LICS*, pages 96–107, 2003.

[Kra97] Markus Kracht. Inessential features. In *Logical Aspects of Computational Linguistics*, volume 1328 of *Lecture Note in Artificial Intelligence*, pages 43–62. Springer-Verlag, 1997.

[Mac86] David B. MacQueen. Using dependent types to express modular structure. In *ACM symposium on Principles of programming languages*, pages 277–286. ACM Press, 1986.

[Mak77] Gennadi S. Makanin. The problem of solvability of equations in a free semigroup. *Mathematics of the USSR Sbornik*, 32, 1977. Translated from Russian.

[Mil99] Robin Milner. *Communicating and mobile systems: the pi-calculus*. Cambridge University Press, 1999.

[Mit84] John C. Mitchell. Coercion and type inference. In *ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 175–185. ACM Press, 1984.

[Mit91] John C. Mitchell. Type inference with simple subtypes. *The Journal of Functional Programming*, 1(3):245–285, 1991.

[Mit96] John C. Mitchell. *Foundations for programming languages*. The MIT Press, 1996.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[MNT98] Martin Müller, Joachim Niehren, and Ralf Treinen. The first-order theory of ordering constraints over feature trees. In *IEEE Symposium on Logic in Computer Science*, pages 432–443. IEEE Computer Society Press, 1998.

[Moz99] Mozart Consortium: Universität des Saarlandes, DFKI, SFB 378, SICS, Université de Louvain. The Mozart system of Oz. Freely available at `www.mozart-oz.org`, 1999.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The definition of standard ML*. MIT Press, 1990.

[MW97] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *ACM SIGPLAN international conference on Functional programming*, pages 136–149. ACM Press, 1997.

[Nau66] Robert Mc Naughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9:521–530, 1966.

[NMT99] Joachim Niehren, Martin Müller, and Jean-Marc Talbot. Entailment of atomic set constraints is PSPACE-complete. In *IEEE Symposium on Logic in Computer Science*, pages 285–294. IEEE Computer Society Press, 1999.

[NP99] Joachim Niehren and Tim Priesnitz. Entailment of non-structural subtype constraints. In *Asian Computing Science Conference*, Lecture Notes in Computer Science 1742, pages 251–265. Springer-Verlag, 1999.

[NP03] Joachim Niehren and Tim Priesnitz. Non-structural subtype entailment in automata theory. *Information and Computation*, 186(2):319–354, 2003.

[OW92] Patrick M. O'Keefe and Mitchell Wand. Type inference for partial types is decidable. In *European symposium on programming*, pages 408–417. Springer-Verlag, 1992.

[Pal99] Adi Palm. Propositional tense logic for trees. In *Mathematics of Language*, pages 74–87. University of Central Florida, 1999.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[Pla99] Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. In *IEEE Symposium on Foundations of Computer Science*, pages 495–500. IEEE Computer Society Press, 1999.

[PO95] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, 1995.

[Pot96] François Pottier. Simplifying subtyping constraints. In *ACM SIGPLAN International Conference on Functional Programming*, pages 122–133. ACM Press, 1996.

[Pot98a] François Pottier. A framework for type inference with subtyping. In *ACM SIGPLAN International Conference on Functional Programming*, pages 228–238. ACM Press, 1998.

[Pot98b] François Pottier. *Type inference in the presence of subtyping: from theory to practice*. PhD thesis, Institut de Recherche d'Informatique et d'Automatique, 1998.

[Pot01] François Pottier. Simplifying subtyping constraints: a theory. *Information and Computation*, 170(2):122–133, 2001.

[Proa] Programming Methods Laboratory, Swiss Institute of Technology Lausanne. The Scala programming system. Freely available at `http://scala.epfl.ch/`.

[Prob] Programming Systems Lab, Universität des Saarlandes. The Alice system. Freely available at `www.ps.uni-sb.de/alice`.

[PS94] Carl Pollard and Ivan A. Sag. *Head-driven phrase structure grammar*. University of Chicago Press, 1994.

[PT96] Vaughan Pratt and Jerzy Tiuryn. Satisfiability of inequalities in a poset. *Fundamenta Informaticae*, 28:165–182, 1996.

[PW78] Mike Paterson and Mark N. Wegman. Linear unification. *Journal of Computer System Sciences*, 16(2):158–167, 1978.

[PWO97] Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.

[Rab69] Michael O. Rabin. Decidability of second-order theories and automata on infinte trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[Reh97] Jakob Rehof. Minimal typings in atomic subtyping. In *ACM Symposium on Principles of Programming Languages*, pages 278–291. ACM Press, 1997.

[Reh98] Jakob Rehof. *The complexity of simple subtyping systems*. PhD thesis, DIKU, Copenhagen, 1998.

[Rey80] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation, Proceedings of a Workshop*, pages 211–258. Springer-Verlag, 1980.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23–41, 1965.

[RV98] Didier Rémy and Jérôme Vouillon. Objective ML: an effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[SAN+] Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. First-order theory of subtyping constraints. *ACM Transactions on Programming Languages and Systems*. To appear.

[SAN+02] Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. First-order theory of subtyping constraints. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 203–216. ACM Press, 2002.

[Sch91] Klaus U. Schulz. Makanin's algorithm for word equations – two improvements and a generalization. In *International Workshop on Word Equations and Related Topics*, Lecture Notes in Computer Science 572, pages 85–150. Springer-Verlag, 1991.

[sE] Open source Erlang. The Erlang programming language. Freely available at `www.erlang.org`.

[Sei94] Helmut Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.

[sf] Python software foundation. The Python programming language. Freely available at `www.python.org`.

[SHC95] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Australian Computer Science Conference*, pages 499–512. Australian Computer Science Communications: Glenelg, South Australia, 1995.

[Spa93] Edith Spaan. *Complexity of modal logics*. PhD thesis, ILLC, University of Amsterdam, 1993.

[Su02] Zhendong Su. *Algorithms for and the complexity of constraint entailment*. PhD thesis, University of California at Berkeley, 2002.

[Tha94] Satish Thatte. Type inference with partial types. *Theoretical Computer Science*, 124:127–148, 1994.

[Tiu92] Jerzy Tiuryn. Subtype inequalities. In *IEEE Symposium on Logic in Computer Science*, pages 308–315. IEEE Computer Society Press, 1992.

[Tiu97] Jerzy Tiuryn. Subtyping over a lattice. In *Computational Logic and Proof Theory, Kurt Gödel Colloquium*, volume 1289 of *Lecture Notes in Computer Science*, pages 84–88. Springer-Verlag, 1997.

[Tre00] Ralf Treinen. Predicate logic and tree automata with tests. In *Foundations of Software and Computer Structures*, Lecture Notes in Computer Science 1784, pages 329–343. Springer-Verlag, 2000.

[TS96] Valery Trifonov and Scott Smith. Subtyping constrained types. In *International Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 349–365. Springer-Verlag, 1996.

[TW69] James W. Thatcher and Jesse B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–82, 1969.

[TW93] Jerzy Tiuryn and Mitchell Wand. Type reconstruction with recursive types and atomic subtyping. In *Theory and Practice of Software Development*, Lecture Notes in Computer Science 668, pages 686–701. Springer-Verlag, 1993.

[VR83] Yu. M. Vazhenin and Bella V. Rozenblat. Decidability of the positive theory of a free countably generated semigroup. In *Mathematics of the USSR Sbornik*, volume 44, pages 109–116, 1983.

[VW86] Moshe Y. Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):183–221, 1986.

[WL99] Pierre Weis and Xavier Leroy. *Le langage Caml*. Dunod, 1999.

[WO89] Mitchell Wand and Patrick O'Keefe. On the complexity of type inference with coercion. In *International conference on Functional programming languages and computer architecture*, pages 293–298. ACM Press, 1989.