Algorithms for and the Complexity of Constraint Entailment

by

Zhendong Patrick Su

B.A. (University of Texas at Austin) 1995
B.S. (University of Texas at Austin) 1995
M.S. (University of California at Berkeley) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Fall 2002

The dissertation of Zhendong Patrick Su is approved:

Chair Date

Date

Date

University of California at Berkeley

Fall 2002

Algorithms for and the Complexity of Constraint Entailment

Abstract

Algorithms for and the Complexity of Constraint Entailment

by

Zhendong Patrick Su
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Alexander S. Aiken, Chair

This thesis attempts to settle some of the longstanding open problems in scalable type systems. In particular, we show that the first-order theory of subtyping constraints is undecidable, and in the case where all type constructors are unary and constants, it is decidable via an automata-theoretic reduction. This automata-theoretic approach is extended to handle the general problem of subtype entailment. Finally, we provide efficient algorithms for simplifying conditional equality constraints, arguably the simplest type language with subtyping.

Professor Alexander S. Aiken
Dissertation Committee Chair

To my wife Yiqing and our daughter Amanda

Contents

List of Figures **v**

List of Tables **vi**

1 Introduction **1**
1.1 Scalable program analysis 1
1.2 Constraint logics 3
1.2.1 Equality constraints 3
1.2.2 Subtyping constraints 4
1.3 Main technical problems 6
1.3.1 Entailment 6
1.3.2 Existential entailment 7
1.3.3 Subtyping constrained types 7
1.4 Thesis contributions 8

2 Background **9**
2.1 Subtype languages and subtyping constraints 9
2.1.1 Types as trees 9
2.1.2 Subtype orders 10
2.1.3 Type variables, expressions, and constraints 12
2.2 Subtype systems 13
2.3 Algorithms for subtype satisfiability 16
2.3.1 Finite structural subtype satisfiability 18
2.3.2 Recursive structural subtype satisfiability 18
2.3.3 Recursive non-structural subtype satisfiability 19
2.3.4 Finite non-structural subtype satisfiability 19
2.3.5 Structural vs. non-structural subtyping 20
2.4 Constraint entailment 21
2.5 Previous results on entailment 23

3 The First-order Theory **25**
3.1 First-order theory of subtyping constraints . . . 25
3.1.1 Entailment is in the ∀-fragment . . . 26
3.1.2 Existential entailment is in the ∀∃-fragment . . . 26
3.1.3 Subtyping constrained types is in the ∀∃-fragment . . . 27
3.2 Undecidability of the first-order theory . . . 28
3.2.1 Representing words as trees . . . 29
3.2.2 Reducing PCP to FOT of subtyping . . . 32
3.2.3 Recursive types . . . 39
3.3 Structural subtyping: a comparison . . . 41

4 Tree Automata and Entailment **43**
4.1 Background on tree automata . . . 43
4.2 Decidability of the monadic fragment . . . 47
4.3 Extending to arbitrary signatures . . . 53
4.3.1 Expressing $x \leq y$ in general . . . 53
4.3.2 Constrained automata for $x = f(y, z)$ . . . 56
4.3.3 Expressing subtype entailment . . . 57
4.3.4 Constrained automata emptiness is undecidable . . . 58
4.4 An example . . . 61

5 Conditional Equality Constraints **66**
5.1 The problem setting . . . 67
5.2 Entailment of equality constraints . . . 70
5.2.1 Simple entailment . . . 71
5.2.2 Restricted entailment . . . 73
5.3 Conditional equality constraints . . . 76
5.3.1 Simple entailment . . . 78
5.3.2 Restricted entailment over atomic constraints . . . 79
5.4 Restricted entailment over conditional equality constraints . . . 84
5.4.1 Closed systems . . . 86
5.4.2 Entailment of pair constraints . . . 91
5.4.3 Reduction of entailment to closed systems . . . 94
5.4.4 Putting everything together . . . 96
5.5 Extended conditional constraints . . . 97
5.5.1 Simple entailment . . . 97
5.5.2 Restricted entailment . . . 99
5.6 Related issues . . . 105

6 Related Work **106**
6.1 Theoretical development . . . 106
6.1.1 Work on type simplification . . . 106

6.1.2 Work on entailment 107
6.1.3 Work on entailment in related domains 108
6.1.4 Scalable type-based program analysis 109
6.2 Practical heuristics 109
6.2.1 Type and constraint simplification 109
6.2.2 Scalable constraint resolution techniques 110

7 Conclusion and Future Work **112**
7.1 Thesis summary 112
7.2 Open problems 113
7.2.1 Non-structural subtype entailment 113
7.2.2 Existential entailment and subtyping constrained types 113
7.2.3 The first-order theory of structural subtyping 114

Bibliography **115**

List of Figures

2.1 The generic subtyping system. 14
2.2 The generic subtyping system formulated with constraints. 15

3.1 Some example representations of words. 30
3.2 Tree prepending example. 32
3.3 A PCP solution viewed as a tree. 33
3.4 The left branch of a solution tree. 34
3.5 Extracting words from a left branch. 36
3.6 Failed attempts for recursive types. 40

5.1 Closure rules for equality constraints. 71
5.2 Congruence closure for equality constraints. 71
5.3 Simple entailment $C \models \alpha = \beta$ over equality constraints. 72
5.4 Restricted entailment $C_1 \models_E C_2$ over equality constraints. 74
5.5 Constructed valuation ρ. 75
5.6 Modified conditional unification algorithm. 76
5.7 Simple entailment $C \models \alpha \Rightarrow \beta$ over conditional equality constraints. . 77
5.8 Example. 81
5.9 Algorithm for restricted entailment over atomic constraints. 82
5.10 Solutions for all subsets of two constraints. 84
5.11 An example constraint system and its closed system. 87
5.12 Example entailment. 94
5.13 Algorithm for simple entailment over extended conditional equality constraints. 99
5.14 Constructed constraint system C_2. 102

List of Tables

5.1 Summary of results. 67

Acknowledgements

I am deeply grateful to many people who directly or indirectly helped me complete this dissertation. First, I want to thank my advisor Alex Aiken for his guidance and support. Alex not only taught me how to do good research, but also provided an excellent example in balancing work and family: being a researcher, a husband, and a father. He is always somebody that I can look up to, and I was very fortunate to have him as a mentor.

When I was an undergraduate student at the University of Texas at Austin, I had the opportunity to learn from two great scholars and teachers Edsger W. Dijkstra and Vladimir Lifschitz. From their teachings and other interactions with them, I gained a deeper appreciation for the joy and excitement of discovering new and "beautiful" knowledge. They made me a better thinker.

I had the opportunity to discuss, with many people, the problems and results reported in this dissertation. In particular, I want to thank Jakob Rehof, Joachim Niehren, Tim Priesnitz, Ralf Treinen, Jens Palsberg, Hubert Comon, Nevin Heintze, Orna Kupferman, and Moshe Vardi for interesting discussions regarding aspects of the problems studied here. I am also grateful to George Necula for useful feedbacks that helped me improve the presentation of this dissertation and to Dorit Hochbaum for her time in reviewing this dissertation.

The graduate student life would have been unbearable without the support and the encouragement of many fellow students. Over the years, I have shared office with a few great fellows, Manuel Fähndrich, Jeff Foster, David Gay, Raph Levien, John Kodumal, and Anders Møller. I thank them for the interesting discussions we had about research, job search, philosophy, and everything else. I also want to thank Ben Liblit, Martin Elsman, Henning Niss, Andy Begel, and members of the Open Source Quality Project for many interesting discussions.

I also want to thank friends in the Chinese For Christ Berkeley Church for providing a relaxing environment for discussions about life and for their caring in difficult times. Without these friends, my years at Berkeley would not have been nearly as enjoyable. In particular, I want to thank our big sister Amy Kao for her constant help, both spiritually and materially.

Last, but not least, special thanks to my family for their constant support. Without them, I would not have been where I am now. I want to thank my wife Yiqing for sharing the difficult times and for her unwavering support and patience, and to thank my daughter Amanda for the immense joy she brought me. I also want to thank my parents and parents-in-law, who are always there to help whenever we need them. Without their help since the birth of Amanda, it would have been difficult for me to concentrate on finishing the research and writing this dissertation.

Zhendong Su
December 2002

Chapter 1

Introduction

This thesis considers algorithmic and complexity issues in designing scalable program analysis. On one hand, we search for efficient algorithms for implementing scalable analyses, and one the other, we show some of the fundamental limitations we face in achieving this goal.

1.1 Scalable program analysis

Program analysis concerns techniques that can be used to discover properties about runtime behaviors of programs. Most program analyses are *static*, meaning that properties are inferred without actually running the program, but simply by looking at the program's source code.

Scalable program analysis is becoming more and more important, because of the tremendous growth in the size and complexity of programs, and because of applications of program analysis such as advanced compiler optimizations [ASU86], software engineering tools [OJ97], and type systems for advanced languages [TS96]. Today, it is challenging to design effective analyses for large software systems.

Many static analyses can be viewed as constraint resolution problems. In such *constraint-based analyses* [NNH99, Aik99], we first generate constraints that describe the relationships among the constructs in a program. Analysis then reduces to finding a "best" solution of these constraints. To analyze large systems, we need to solve the constraints efficiently.

Constraint-based program analysis is attractive in part because a number of systems developed in this style seem to scale very well. Examples of this class of analysis include type systems for functional programming languages such as ML [MTH90] and for object-oriented languages such as Java [GJS96] (a detailed example is given in Chapter 2). Most of these analyses are *syntax-directed*, meaning that the constraints that describe the data- or control-flow of a program are gathered by a simple recursive traversal of the abstract syntax tree representation of the program being analyzed. After generation, the constraints are fed into a constraint resolution engine for processing to get the desired information about the program—for example, whether there is a type error or whether the value of a particular variable is constant.

Since the generated constraints are usually linear in the size of the program being analyzed, scalability of the analyses depends primarily on the efficiency of constraint resolution. At this point one can reasonably argue that we have a rather good understanding of the various satisfaction problems that arise in program analysis and type systems [AWL94, And94, FFK+96, Hei94, JM79, MW97, PS91, Rey69, Shi88, FF97, FFSA98, SFA00].

However the problem of *constraint simplification*, that is, replacing one constraint set with a simpler but equivalent one, is not at all well understood. Constraint simplification arises naturally in polymorphic subtyping systems, which bring together parametric polymorphism (as in ML [Mil78]) with subtyping (as in object-oriented languages such as Java [GJS96]). In such type systems, we have *polymorphically constrained types*, polymorphic types that are constrained by a set of subtyping constraints. For such a type system to be practical, scalable, and understandable to the user, it is important to simplify the constraints associated with a type.

Another area where constraint simplification arises is *polymorphic program analyses* with constraints, in which a function (or module) is analyzed, and information about this function is summarized with a constraint set. At different call sites of the function, the constraint set is instantiated with fresh constraint variables and duplicated. This duplication of the constraint set provides better analysis results by distinguishing the different call sites (in contrast to *monomorphic* analyses, in which a single constraint set is used at all the call sites). For these polymorphic analyses to be practical, simplifying the constraint set is important.

There are two facets to the problem of simplifying constraints. First, the duplication of the constraints is a costly step. Recent work [RF01] address this issue by applying ideas from interprocedural dataflow analysis [RHS95] to polymorphic label-flow in type-based program analysis [Mos96]. Instead of copying the constraints themselves, we only need to *remember* how the instance variables are instantiated. This method may be viewed as an alternative and better implementation of the duplication of the generic constraints.

The second problem we face is that the original constraint set to be copied can be large and complicated. This problem is orthogonal to the first problem. It is beneficial to replace a large constraint system with an equivalent and smaller one, provided the computation needed to perform this simplification is not itself too expensive. The constraints can be more efficiently manipulated when they are simple. In addition, in terms of presenting a polymorphically constrained type to the programmer, it needs to be simplified as much as possible for better understanding and easier manipulation. This second problem is what we consider in this thesis.

1.2 Constraint logics

Constraint formalisms are the vehicles for specifying particular program analyses and types systems. They influence both how an analysis should be designed and how efficiently an analysis can be realized.

In this section, we briefly discuss two popular constraint formalisms for expressing program analyses and type systems. Additional background information is given in Chapter 2.

1.2.1 Equality constraints

Equality constraints have been widely used in two areas, logic programming [Llo87] and static type systems, such as the Hindley/Milner style type systems [PW78].

Equality constraints are popular in part because they are easily understood and efficient algorithms exist for solving equality constraints [PW78]. A slightly less efficient but simpler implementation can be obtained using the union/find data structure [Tar75]. Many program analyses can be formulated as unification problems based on equality constraints [PW78, Das00, Ste96, EHM$^+$99, Hen88, PL99].

One problem with equality constraints is the undirected nature of how information flows, because the computation is based on computing equivalence classes. Thus, information about a term is merged with the information of all the terms that are "reachable" from this particular term. Usually, analyses based on equality constraints are not very precise, which often becomes especially evident when applied to large problems. To address this problem, many analyses support directed flow of information. This added expressiveness leads to the notion of subtyping, which we discuss next.

1.2.2 Subtyping constraints

Many programming languages have some form of subtyping. The most common use is in the sub-classing mechanisms in object-oriented languages such as Java [GJS96] and C++ [Str95]. Also common is the notion of "coercion" [Mit84], for example automatic conversion from integers to floating point numbers. Subtyping constraints of the form $\tau_1 \leq \tau_2$ are used to capture that the type τ_1 is a *subtype* of τ_2. For example, the constraint **int** $\leq$ **real** means that at any place a floating point number is expected, an integer can be used instead. Notice that with equality constraints, this would require that the two types are the same, and it becomes impossible to pass an argument of type **int** to a function expecting a **real**.

Since the original results of Mitchell [Mit91], type checking and type inference for subtyping systems have received steadily increasing attention. The primary motivations for studying these systems today are program analysis algorithms based on subtyping (see, for example, [AWL94, And94, FFK$^+$96, Hei94, MW97, PS91, Shi88]) and, more speculatively, richer designs for typed languages ([OW97]).

Types in subtype systems are typically interpreted over trees over some base elements (drawn from a finite lattice or a partial order [DP90]). The trees can be infinite if recursive types are allowed. There are two choices for the subtype relation. In a system with *structural subtyping* only types with the same shape are related, and they are related by some additional structural rules besides the subtype relation of the base elements. In a system with *non-structural subtyping*, there is a "least" type $\perp$ and a "largest" type $\top$. Types are related by the same set of rules as in the structural case with the additional rules that $\perp$ is smaller than any type and $\top$ is larger than any type. A more detailed introduction to structural and non-structural subtyping is given in Chapter 2.

In a subtype system (or in any program analysis based on the notion of subtyping), the subtyping relation dictates an order on the types, which in turn provides a notion of "directional flow of information". This added power of subtyping is much more expressive than equality, and at the same time, much more subtle. With this directivity, instead of an equivalence computation as for equality constraints, we have a "reachability" computation. This leads to some very interesting and intriguing problems, as we will discuss next.

Subtyping algorithms invariably involve systems of subtype constraints $\tau_1 \leq \tau_2$, where the τ_i are types that may contain type variables. There are two interesting questions we can ask about a system of subtyping constraints C:

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

For (1), the basic algorithms for solving many natural forms of subtyping constraints are by now quite well understood (e.g., see [Reh98]). For (2), there has been much less progress on subtype entailment, although entailment is as important as constraint resolution in applications of subtyping. For example, a type-based program analysis extracts some system of constraints C from a program text; these constraints are the model of whatever program property is being analyzed. A client of the analysis (e.g., a program optimization system) interacts with the analysis by asking queries: Does a particular constraint $\tau_1 \leq \tau_2$ hold in C? Or in other words, does C entail $\tau_1 \leq \tau_2$? As another example, in designing a language with expressive subtyping relationships, checking type interfaces also reduces to a subtype entailment problem. While no mainstream language has such expressive power today, language researchers have encountered just this problem in designing languages that blend ML-style polymorphism with object-oriented style subtyping, which leads to *polymorphically constrained types* (see, again, discussion in [OW97]).

1.3 Main technical problems

In this section, we give an overview of the main problems that we consider in this thesis. More background material is discussed in Chapter 2.

Our focus is on algorithms for designing scalable and expressive type systems and static program analysis. There are three fundamental problems that are closely related to constraint simplification and are the basic building blocks for designing expressive type systems and powerful program analyses.

Corresponding to polymorphic type schemes in Hindley/Milner style type systems, polymorphic subtype systems have so-called *polymorphically constrained types* (also known as *constrained types*), in which a type is restricted by a system of constraints [AW93, TS96, AWP97]. An ML style polymorphic type can be viewed as a constrained type with no constraints. For example,

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

is a constrained type: a function type $\alpha \to \beta$ restricted by the constraints $\{\alpha \leq \mathbf{int} \to \mathbf{int}, \mathbf{int} \to \alpha \leq \beta\}$. Let $\tau \backslash C$ be a constrained type, and let ρ be a satisfying valuation for C. The type $\rho(\tau)$ is called an *instance* of $\tau \backslash C$.

1.3.1 Entailment

In practice, constrained types can be large and complicated. Thus it is important to simplify the types [Pot96, MW97, FA96] to make the types and the associated constraints smaller. Type and constraint simplification is related to the following decision problem of *constraint entailment*: A constraint system C *entails* a constraint c written $C \vDash c$, if every solution of C also satisfies the constraint c.

1.3.2 Existential entailment

Let C_1 and C_2 be two constraint systems and E be a finite set of variables. For convenience, in the following discussions, we use $\exists E$ as a shorthand for $\exists x_1, \ldots, \exists x_n$, where $E = \{x_1, \ldots, x_n\}$. For a constraint system C, the type variables in C are called the *free variables* of C, denoted by $\mathrm{fv}(C)$.

The notion of *existential entailment*, written $C_1 \vDash \exists E.C_2$, is a more powerful notion of entailment.[1] The entailment holds if for every solution of C_1, there exists a solution C_2 such that both solutions coincide on variables $\mathrm{fv}(C_2) \setminus E$ (We assume w.l.o.g. that $\mathrm{fv}(C_1) \cap E = \emptyset$). This notion is interesting because usually for a constrained type, we are only interested in variables appearing in the type, and there are often many "internal" variables in the constraints we may wish to eliminate. This notion of entailment allows more powerful simplification and is likely to be more expensive.

1.3.3 Subtyping constrained types

In polymorphic subtype systems, we may need to determine whether one constrained type is a subtype of another constrained type [TS96]. Let $\tau_1 \backslash C_1$ and $\tau_2 \backslash C_2$ be two constrained types. We wish to check whether $\tau_1 \backslash C_1 \leq \tau_2 \backslash C_2$ which holds if for every instance τ_2' of $\tau_2 \backslash C_2$, there exists an instance τ_1' of $\tau_1 \backslash C_1$ such that $\tau_1' \leq \tau_2'$.

Although extensive research has been directed at these problems [Reh98, HR97, HR98,FF97,NP99,NP01,TS96,AWP97,FA96,Pot96,MW97,Pot01], their decidability and exact complexity have been open for many years. In this thesis, we propose new methods for attacking these open problems. We show that a more general problem is undecidable, which provides some technical evidence that some of these problems

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

might be undecidable. In addition, the new techniques introduced are used to settle the decidability of some interesting fragments of these problems.

1.4 Thesis contributions

The main topic of this thesis is the study of a general theory of subtyping constraints, which embodies the various forms of the subtype entailment problem. We hope that with this general theory, one can gain further understanding about the various open entailment problems under the non-structural type order.

Here are the contributions of the thesis:

- We show that the first-order theory of subtyping constraints is undecidable (Chapter 3). This result suggests that open problems involving entailment over non-structural subtyping might in fact be undecidable. The result holds for any type language that includes a bottom element $\perp$ (or a top element $\top$) and any binary or greater arity type constructor.

- In the case where all type constructors are unary and constants, we show that the first-order theory is decidable via an automata-theoretic reduction (Chapter 4). Furthermore, we show how to extend this proof technique to handle subtyping entailment under an arbitrary type signature. We conjecture that this automata-theoretic approach may be useful in resolving other entailment problems.

- Finally, inspired by practical applications, we strengthen the non-structural type relation (see Chapter 5 for details). Under this restricted non-structural type order, we devise novel polynomial time algorithms for entailment and existential entailment (Chapter 5).

Chapter 2

Background

In this chapter, we review basic concepts and results from the literature on subtype systems. We define the various choices of type expression languages, type orders, and the notion of subtyping constraints. We then review previous results on satisfiability of subtyping constraints interpreted over these choices of subtyping languages. Finally we survey previous results on constraint simplification and entailment.

2.1 Subtype languages and subtyping constraints

Subtyping systems are generalizations of the usual equality-based type systems such as the Hindley/Milner type system of ML [Mil78]. Before discussing any type system, we first need to discuss its type language. There are two components to a subtype language: how to form a type and how to order the types.

2.1.1 Types as trees

Types can be viewed as trees built from some regular tree grammar [GS84]. For the purpose of this thesis, we take this syntactic (or operational) view in defining our type languages and their interpretations. We are not concerned with domain-theoretic interpretations of types, such as with ideals [MPS86]. Please refer to [Mit96] for more information on these topics.

Let Σ be a ranked alphabet. Elements in the alphabet are called *constructors* in the type language. Each constructor f has an associated *arity*, indicating the number of arguments that f expects. The set of constructors with arity n is denoted Σ_n. Type constructors with arity zero are *type constants*, and they form the basis of a type language. We consider types as *trees* following [KPS93].

Definition 2.1.1 (Tree) A *tree* t over a ranked alphabet Σ is a mapping from a prefix-closed set $\text{pos}(t) \subseteq \mathbb{N}^*$ into Σ. The set of *positions* pos of t satisfies:

- $\text{pos}(t)$ is nonempty and prefix-closed;
- for each $\pi \in \text{pos}(t)$, if $t(\pi) \in \Sigma_n$, then $\{i \mid \pi i \in \text{pos}(t)\} = \{1, \ldots, n\}$.

Intuitively, the set of positions $pos(t)$ defines the structure of the tree t, and each position is labeled with an element from the alphabet.

A tree t is *finite* (resp. *infinite*) if $\text{pos}(t)$ is a finite (resp. infinite) set.

Simple types [Mit91] are interpreted over finite trees, while *recursive types* [AC91] are interpreted over *regular trees*, which are possibly infinite trees with finitely many sub-terms.

Example 1 (Types) Consider the type language generated by the alphabet:

$$\{\bot, \top, \rightarrow\}$$

where $\rightarrow$ is the usual function type constructor, and $\bot$ and $\top$ are two type constants.

An example simple type is $(\bot \rightarrow \top) \rightarrow \bot$. An example recursive type, written in the μ-term notation, is $\mu\alpha.\alpha \rightarrow \bot$.

Recursive types are usually specified with the fixpoint notation: $\mu\alpha.\tau$, and, are also called μ-terms [Pie02].

2.1.2 Subtype orders

In a subtype system, we also need to decide how to order the types, *i.e.*, to specify a relation among the types. One can view an equality-based type system such as that for ML [Mil78] as an instance of a subtype system. In an equality-based type system, all the types are put in a discrete partial order [DP90].

Two subtype orders arise naturally in practice: the *structural subtype order* and the *non-structural subtype order*. In discussing these type orders, we work with a type language generated from the alphabet:

$$\Sigma = B \mid \times \mid \rightarrow$$

where B is the set of type constants and $\times$ and $\rightarrow$ are two binary constructors for building product types (or records) and function types. The types are specified by the following grammar:

$$\tau = B \mid \tau \times \tau \mid \tau \rightarrow \tau$$

Structural subtype order

In the structural type order, the type constants B form the basis of the type order. The constants are put in a partial order $(B, \leq)$. We can then lift the order $\leq$ to work on all types with the following structural rules:

- $\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'$ iff $\tau_1 \leq \tau_1'$ and $\tau_2 \leq \tau_2'$, for any types τ_1, τ_2, τ_1', and τ_2';
- $\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'$ iff $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$, for any types τ_1, τ_2, τ_1', and τ_2'.

The resulting order on the types is again a partial order. In practice, we assume that $(B, \leq)$ forms a lattice. Thus the derived order on all the types also forms a lattice. This restriction is purely for the purpose of efficiency for the problems associated with a subtype system. If we allow general partial orders, even satisfiability becomes PSPACE-hard [Tiu92].

Non-structural subtype order

Notice that for the structural order, two types are related only if they have exactly the same shape. In the non-structural type order, this restriction is lifted. Two types may be in the subtype relation even if they do not have the same shape. This notion of type order is arguably more popular and natural than the structural type order, and it is used in many subtype systems and subtype-based program analysis.

In the non-structural type order, two distinguished constants are added to the type language, a *smallest type* $\bot$ and a *largest type* $\top$. The revised type language is given by:

$$\tau = \bot \mid \top \mid B \mid \tau \times \tau \mid \tau \to \tau$$

The non-structural type order is given by:

- $\bot \leq \tau \leq \top$, for any τ;
- $\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'$ iff $\tau_1 \leq \tau_1'$ and $\tau_2 \leq \tau_2'$, for any types τ_1, τ_2, τ_1', and τ_2';
- $\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$ iff $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$, for any types τ_1, τ_2, τ_1', and τ_2'.

Besides the structural rules, two rules are added, which essentially say that $\bot$ is smaller than any type and $\top$ is larger is any type.

2.1.3 Type variables, expressions, and constraints

In a type inference system, we may not know all types at all times. In the case where the exact type for an expression is not known, we use a *type variable* to denote the type of such an expression. The exact type of the expression is determined later. We assume that there are a denumerable set of type variables $\mathcal{V}$. The complete type expression language is given by the following grammar:

$$\tau = \mathcal{V} \mid \bot \mid \top \mid B \mid \tau \times \tau \mid \tau \to \tau$$

We write $T(\Sigma)$ to denote the set of finite *ground types* (types without variables), where Σ is the alphabet:

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

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

A *subtype constraint* is an inequality of the form $\tau_1 \leq \tau_2$ for type expressions τ_1 and τ_2. A *subtype constraint system* is a conjunction of a finite set of subtyping constraints. When it is clear from the context, we drop the word "subtype" and simply say a constraint or a constraint system. For a constraint system C, the type variables in C are called the *free variables* of C, denoted by $\mathrm{fv}(C)$.

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

- $\rho(\bot) = \bot$;
- $\rho(\top) = \top$;
- $\rho(\tau_1 \to \tau_2) = \rho(\tau_1) \to \rho(\tau_2)$;
- $\rho(\tau_1 \times \tau_2) = \rho(\tau_1) \times \rho(\tau_2)$.

A valuation ρ *satisfies* a constraint $\tau_1 \leq \tau_2$, written $\rho \vDash \tau_1 \leq \tau_2$ if $\rho(\tau_1) \leq \rho(\tau_2)$ holds in the lattice $T(\Sigma)$. A valuation ρ *satisfies* a constraint system C, written $\rho \vDash C$, if ρ satisfies all the constraints in C. A constraint system C is *satisfiable* if there is a valuation ρ such that $\rho \vDash C$. The set of valuations satisfying a constraint system C is the *solution set* of C, denoted by $S(C)$. We denote by $S(C)|_E$ the set of solutions of C restricted to a set of variables E by projecting the solutions on those variables in E only. The *satisfiability problem* for a constraint language is to decide whether a given system of constraints is satisfiable. It is well-known that the satisfiability of a subtype constraint system can be decided in polynomial time by a test for *consistency* of the given constraint set according to a set of syntactic rules when the type constants B form a lattice [PO95a, Pot96, KPS94]. More information on checking satisfiability of constraint systems is given in Section 2.3.

2.2 Subtype systems

In this section, we introduce a generic subtype system [Mit91, Mit84] for λ-calculus [Bar91] to illustrate the relationship between type systems, and satisfiability and entailment of typing constraints. This type system extends the simply typed λ-calculus [Hin97] by adding a rule of *subsumption.*

$$\frac{}{A \cup \{x : \tau\} \vdash x : \tau}\ (\mathsf{var})$$

$$\frac{A \cup \{x : \tau_1\} \vdash e : \tau_2}{A \vdash \lambda x : \tau_1 . e : \tau_1 \rightarrow \tau_2}\ (\mathsf{abs})$$

$$\frac{A \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad A \vdash e_2 : \tau_1}{A \vdash e_1 e_2 : \tau_2}\ (\mathsf{app})$$

$$\frac{A \vdash e : \tau_1 \qquad \tau_1 \leq \tau_2}{A \vdash e : \tau_2}\ (\mathsf{sub})$$

Figure 2.1: The generic subtyping system.

The type system is given in Figure 2.1, and it is given in the style of a type checking system [Car96, CW85]. There are four type rules in this type system. Each rule consists of a logical judgment of the form:

$$A \vdash e : \tau$$

which reads "under the type environment A, expression e has the type τ". Here A is a *type environment* that consists of elements of the form $x : \tau$ (unique for each variable x) to specify the type bindings for λ-bound variables. The first three rules (var, abs, and app) are precisely those for the simply typed λ-calculus [Hin97]. The var rule is used to retrieve the type of a variable x from the environment A. The abs and app rules are for typing λ-abstractions and applications. These are all standard. The rule that is unique to a subtype system is the sub rule. It says that if an expression has type τ, then it should also has the type of any supertype of τ.

We say that a judgment is *derivable* if it is provable in the system. An expression e has the type τ within this type system if the judgment $\vdash e : \tau$ if derivable, in which case, we say that e is *typable*.

We give next an inference version of the type system in Figure 2.1 to show the connection between typability and satisfiability of typing constraints. The system is given in Figure 2.2. In this system, we build subsumption into the other three rules.

$$\frac{}{A \cup \{x : \tau\} \vdash x : \tau, \{\}} \ (\mathsf{var})$$

$$\frac{A \cup \{x : \tau_1\} \vdash e : \tau_2, C}{A \vdash \lambda x.e : \alpha, C \cup \{\tau_1 \to \tau_2 \leq \alpha\}} \ (\mathsf{abs})$$

$$\frac{A \vdash e_1 : \tau_1, C_1 \qquad A \vdash e_2 : \tau_2, C_2}{A \vdash e_1 e_2 : \alpha, C_1 \cup C_2 \cup \{\tau_1 \leq \tau_2 \to \alpha\}} \ (\mathsf{app})$$

Figure 2.2: The generic subtyping system formulated with constraints.

The judgments are now of the form:

$$A \vdash e : \tau, C$$

with an additional element C at the end. It reads "under the type environment A, the expression e has type τ if the constraints C have a solution". The constraint set C is accumulated from the sub-expressions to produce a global constraint set. In the rules, the type variable α is *fresh*, meaning not already in use. The var rule is the same as before, except with an empty constraint set. In the abs rule, a subsumption step is inserted at the end for $\lambda x.e$. In the app rule, a subsumption step is inserted for the judgment $A \vdash e : \tau_1, C_1$ to get:

$$A \vdash e : \tau_2 \to \alpha, C_1 \cup \{\tau_1 \leq \tau_2 \to \alpha\}$$

It is a standard result that an expression is typable in the type checking system if and only if the typing constraints in the type inference system are satisfiable. In addition, each solution of the constraints corresponds to a valid type derivation for that expression.

Instantiating with different type structures, we get specific instances of the generic subtype system. Mitchell introduced and studied finite structural subtyping [Mit91], which was subsequently studied by Fuh and Mishra [FM90,FM89] and others. Tiuryn and Wand [TW93], among others, have studied the natural generalization of finite structural subtyping to structural recursive subtyping. Amadio and Cardelli [AC91, AC93] introduced non-structural recursive subtyping. The type inference problem for non-structural subtyping has been studied extensively [KPS94,KPS93,PO95a,OW92, PWO97].

2.3 Algorithms for subtype satisfiability

In this section, we recall previous work on checking the satisfiability of subtype constraints. These algorithms are all based on the idea of checking consistency in the closure of the constraints w.r.t. some closure rules. This is similar to the use of unification closure in solving equality constraints over first-order terms.

In this section, we assume that the base elements of the subtype language form a lattice, instead of an arbitrary partial order. If arbitrary partial orders are allowed, it becomes PSPACE-hard to check satisfiability of subtype constraints under the structural subtype order over both finite and recursive type trees [Tiu92]. Even in the special case of *atomic satisfiability*, where all constraints are between *atoms*—variables and constants, satisfiability is NP-hard [PT96]. Atomic satisfiability can be checked in linear time if lattices are used instead [RM99]. Thus, it becomes necessary to restrict our attention to lattices only for the purpose of efficiency.

Before describing any algorithms, we first define *weak unifiability* and *constraint closure*. We follow [Reh98] in the following discussions, and more information can be found in [Reh98].

Definition 2.3.1 (Weak unifiability) Let C be a constraint set, and $\star$ be an arbitrary and fixed constant. For any type expression τ, $\tau^\star$ denotes the same type expression as τ except all constants in τ are replaced with $\star$. Define the constraint set E_C by:

$$E_C \overset{\text{def}}{=} \{\tau_1^\star = \tau_2^\star \mid \tau_1 \leq \tau_2 \in C\}$$

The constraint set C is called *weakly unifiable* if and only if E_C is unifiable. We require an occurs-check [ASU86, PW78] for the case of unifiability over finite types.

Definition 2.3.2 (Constraint closure) Let C be a constraint set. We say that C is *closed* if the following conditions are satisfied:

- (transitivity)
  $\tau_1 \leq \tau_2 \in C, \tau_2 \leq \tau_3 \in C$ implies that $\tau_1 \leq \tau_3 \in C$;
- (product decomposition)
  $\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2' \in C$ implies that $\tau_1 \leq \tau_1' \in C$ and $\tau_2 \leq \tau_2' \in C$.
- (function decomposition)
  $\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2' \in C$ implies that $\tau_1' \leq \tau_1 \in C$ and $\tau_2 \leq \tau_2' \in C$.

We next define the notion of consistency for a closed constraint set. The definitions differ slightly for the structural and the non-structural subtype orders. We discuss them separately.

Definition 2.3.3 (Structural consistency) Let C be a closed constraint set. We call C *ground consistent* if for all constants a and b, if $a \leq b \in C$, then $a \leq b$ holds in the base lattice of constants. A closed constraint set C is called *structurally consistent* if C is both weakly unifiable and ground consistent.

Definition 2.3.4 (Non-structural consistency) Let C be a closed constraint set. We call C *non-structurally consistent* if for all $\tau_1 \leq \tau_2 \in C$, one of the following conditions is satisfied:

- $\tau_1 = \bot$;
- $\tau_2 = \top$;
- either τ_1 or τ_2 is a variable;
- the top-level constructors of τ_1 and τ_2 are the same, *i.e.*, either $\tau_1 = \sigma_1 \times \sigma_1'$ and $\tau_2 = \sigma_2 \times \sigma_2'$ or $\tau_1 = \sigma_1 \rightarrow \sigma_1'$ and $\tau_2 = \sigma_2 \rightarrow \sigma_2'$ for some type expressions σ_1, σ_1', σ_2, and σ_2'.

Next, we briefly discuss how to check satisfiability of subtype constraints interpreted over various choices of the type lattice.

2.3.1 Finite structural subtype satisfiability

In the case of finite structural subtype order, Tiuryn [Tiu92] shows that satisfiability can be decided in polynomial time. The technique, as for all the other variants, is to show that satisfiability is equivalent to structural consistency. Structural consistency can be decided in two steps. First, we check for weak unifiability, which can be done in linear time [PW78]. Second, we compute the closure of the constraints, which can be done in cubic time with a dynamic transitive closure computation [Yel93]. Finally, checking for ground consistency can be done in quadratic time, because the size of the closure of the constraints is worst-case quadratic in the size of the original constraints.

Theorem 2.3.5 (Tiuryn [Tiu92]) A constraint set C is satisfiable in the type structure of finite structural types if and only if C is weakly unifiable and ground consistent. In particular, satisfiability with structural, finite subtyping over a lattice is in PTIME.

This case, as pointed out by Rehof in his thesis [Reh98], can be reduced to matrix multiplication, which can be performed in sub-cubic time [Str69]. Thus, structural, finite subtyping does not suffer from the "cubic bottleneck" problem [HM97b,MR97].

2.3.2 Recursive structural subtype satisfiability

In the case where recursive types are allowed, a theorem similar to that of Theorem 2.3.5 can be stated.

Theorem 2.3.6 A constraint set C is satisfiable in the type structure of recursive structural types if and only if C is weakly unifiable and ground consistent. In particular, satisfiability with recursive structural subtyping over a lattice is in PTIME.

This theorem immediately suggests a cubic time algorithm for checking recursive structural subtype satisfiability. First, checking weak unifiability requires almost linear time. Second, computing constraint closure takes cubic time. Finally, checking ground consistency takes quadratic time. In this case, no sub-cubic time algorithm is known for computing the constraint closure.

2.3.3 Recursive non-structural subtype satisfiability

Because the case of recursive non-structural subtype satisfiability is simpler than that for the finite case, we discuss the recursive case first.

It is shown by Palsberg and O'Keefe [PO95b] and Pottier [Pot96] that satisfiability in this case can also be decided in PTIME. In particular, the problem is again shown to be equivalent to consistency of the constraints. Because consistency can be checked in cubic time again by reducing to dynamic transitive closure [Yel93], the problem can, in fact, be solved in cubic time.

Theorem 2.3.7 (Palsberg & O'Keefe [PO95b], Pottier [Pot96]) A constraint set C is satisfiable in the type structure of recursive non-structural types if and only if C is non-structurally consistent. In particular, recursive non-structural satisfiability is in PTIME, and can be decided in cubic time.

In fact, by following MacQueen, Plotkin, and Sethi [MPS86] and subsequently applied by Aiken and Wimmers [AW93], we can view the type structure as a complete metric space [Apo74]. The satisfiability, and in fact, finding all solutions of the constraints can be reduced to the problem of solving contractive equations in this metric space. We first compute the closure of the constraints. If the constraints are consistent, we inductively order the variables, and reduce the constraints to a set of contractive equations [AW93, AW92]. By Banach's Fixpoint Theorem, contractive equations have solutions [MPS86, Apo74]. Thus, this gives another proof that satisfiability is equivalent to consistency.

The algorithms given in [PO95b, Pot96] do not appear to handle more than one non-trivial type constructor. As sketched here, the algorithms work for an arbitrary type signature.

2.3.4 Finite non-structural subtype satisfiability

Kozen, Palsberg, and Schwartzbach [KPS94] show that satisfiability for partial types [OW92, Tha94], that is, types without either $\top$ or $\bot$, but not both, can be decided in cubic time both for finite and recursive types. Palsberg, Wand, and

O'Keefe [PWO97] extend this work to show that finite non-structural subtyping satisfiability can be decided in cubic time as well. Here is the basic idea. First, we check for consistency of the constraints, which can be done in cubic time. Then, from the closure of the constraints, we construct a special kind of automaton that has quadratic number of states and cubic number of transitions. Finally we search for a cycle in the constructed automaton. If the automaton is acyclic, then the constraints have a finite solution. Otherwise, the constraints have only regular solutions.

Theorem 2.3.8 (Palsberg, Wand, & O'Keefe [PWO97]) It is decidable in cubic time whether a constraint set has a finite solution over non-structural subtyping.

2.3.5 Structural vs. non-structural subtyping

There are some technical differences between structural and non-structural subtyping in the complexity of the satisfiability and entailment problems.

The first separation is at the level of checking for satisfiability. Finite structural subtyping constraints can be solved in sub-cubic time [Reh98] by a reduction to matrix multiplication [Str69] through stratification (that is, by solving a separate problem for each level of the types). Since recursive types cannot be stratified into levels, it is open (and appears unlikely) whether this method can be extended to structural recursive subtyping. Satisfiability of non-structural subtyping constraints can be decided in cubic time over both finite and recursive type structures.

There are two independent approaches [HM97b, MR97] that attempt to characterize the difficulty of devising sub-cubic time algorithms for certain dataflow and control-flow analyses [Shi88, PS91, And94, AC91, PO95b], which are equivalent to satisfiability problem for non-structural subtyping constraints. Both approaches reduce to or show equivalence of some well-known hard problems to these dataflow or control-flow analysis problems. Heintze and McAllister [HM97b] consider the class of 2NPDA, those problems that can be solved with a two-way nondeterministic pushdown automata [HU79]. It was shown, in 1968, that any problem in the class of 2NPDA can be solved in cubic time [AHU68], but no sub-cubic time algorithm exists for any arbitrary 2NPDA problem. Heintze and McAllister show that control-flow analysis is, in fact, 2NPDA-complete, which means that if there is a sub-cubic time algorithm for the standard control-flow analysis, then any problem in the class of 2NPDA can also be solved in sub-cubic time. Melski and Reps [MR97]) consider the problem of CFL graph reachability, which has been used extensively in formulating many program analysis problems [Rep98, HRB88, Cal88]. They show the intercovertibility of CFL reachability problems and set-based analysis [Hei92], a certain type of dataflow analysis.

As we will see in the next section, there are also some results at the level of entailment (a problem that we introduce in the next section) for various choices of the type structure that separate structural and non-structural subtyping (Section 2.4). Finite structural subtyping entailment is coNP-complete [HR97] and finite non-structural subtyping entailment is PSPACE-hard [HR98]. Both structural recursive subtyping entailment and non-structural recursive subtyping entailment are PSPACE-hard [HR98]. In the class of structural recursive subtyping entailment, the problem is known to be in PSPACE, and thus PSPACE-complete [HR98]. However, for non-structural subtyping, even the decidability of entailment is open.

2.4 Constraint entailment

We give here the formal statement of the problems that we consider in this thesis. We state the problems in general terms, where the various unspecified parameters can be instantiated accordingly in later chapters. We mainly consider two fundamental entailment problems.

Definition 2.4.1 (Entailment) Let C_1 and C_2 be two constraint sets. We say that C_1 *entails* C_2, written $C_1 \vDash C_2$ if for all valuations ρ, if $\rho \vDash C_1$, then it also holds that $\rho \vDash C_2$.

Definition 2.4.2 (Existential entailment) Let C_1 and C_2 be two constraint sets and E a set of variables. We say that C_1 *existentially entails* C_2, written $C_1 \vDash \exists E.C_2$, if for every valuation $\rho \vDash C_1$, there exists a valuation $\rho' \vDash C_2$ such that ρ and ρ' agree on variables $\mathrm{fv}(C_2) \setminus E$.

Existential entailment is a more powerful notion than entailment. In the literature, the term *restricted entailment* is also used, which is written $C_1 \vDash_{E'} C_2$, where $E' = \mathrm{fv}(C_2) \setminus E$. Notice that, in the above definition, we can assume, w.l.o.g., that $\mathrm{fv}(C_1) \cap E = \emptyset$. Existential entailment is interesting because usually for a constrained type, we are only interested in variables appearing in the type, and there are often many "internal" variables in the constraints we may wish to eliminate. This notion of entailment allows more powerful simplification and is likely to be more expensive.

As an example, consider the following simple constraint set:

$$C = \{\alpha \leq \beta, \beta \leq \gamma\}$$

If β is an internal variable, then C is equivalent to the following:

$$C' = \{\alpha \leq \gamma\}$$

which is justified by the fact that both of the following entailments hold:

$$C \vDash \exists\beta.C' \text{ and } C' \vDash \exists\beta.C$$

However, although $C \vDash C'$, but $C' \nvDash C$. Thus, the equivalence of C and C' cannot be established with entailment.

In Hindley/Milner style type systems, a `let`-bound expression can have different types in different contexts. This type of parametric polymorphism is obtained through polymorphic type schemes [Mil78]. In polymorphic subtype systems, there is the similar notion of a constrained type, in which a type is restricted by a system of subtyping constraints [AW93, TS96, AWP97]. An ML style polymorphic type can be viewed as a degenerated constrained type, where there are no constraints to restrict the type.

Definition 2.4.3 (Constrained types) A *constrained type* $\tau \backslash C$ is a type τ restricted by a constraint set C. For a constrained type $\tau \backslash C$, let ρ be a satisfying valuation for C, the ground type $\rho(\tau)$ is an *instance* of $\tau \backslash C$.

For example,

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

is a constrained type. By mapping α to $\mathbf{int} \to \mathbf{int}$ and β to $\mathbf{int} \to (\mathbf{int} \to \mathbf{int})$, we get an instance for this constrained type:

$$(\mathbf{int} \to \mathbf{int}) \to (\mathbf{int} \to (\mathbf{int} \to \mathbf{int}))$$

In polymorphic subtype systems, we may need to determine whether one constrained type is a subtype of another constrained type [TS96].

Definition 2.4.4 (Subtyping constrained types) Let $\tau_1\backslash C_1$ and $\tau_2\backslash C_2$ be two constrained types. We say that $\tau_1\backslash C_1 \leq \tau_2\backslash C_2$ if for every instance τ_2' of $\tau_2\backslash C_2$, there exists an instance τ_1' of $\tau_1\backslash C_1$ such that $\tau_1' \leq \tau_2'$.

We can assume, w.l.o.g., that C_1 and C_2 do not have any variables in common. In addition, we can restrict τ_1 and τ_2 to variables because

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

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

2.5 Previous results on entailment

In this section, we give an overview of prior results on subtyping entailment and simplification.

Henglein and Rehof give the first systematic study of the complexity of subtyping entailment [HR97, HR98, Reh98]. They completely characterize the complexity of entailment for structural subtyping.

Theorem 2.5.1 (Henglein & Rehof [HR97]) Finite structural subtype entailment is coNP-complete.

Theorem 2.5.2 (Henglein & Rehof [HR98]) Recursive structural subtype entailment is PSPACE-complete.

For non-structural subtype entailment, while the decidability is still open, there are lower bounds on the complexity of the problems given again by Henglein and Rehof [HR98].

Theorem 2.5.3 (Henglein & Rehof [HR98]) Finite non-structural subtype entailment is PSPACE-hard.

Theorem 2.5.4 (Henglein & Rehof [HR98]) Recursive non-structural subtype entailment is PSPACE-hard.

Niehren and Priesnitz consider the problem of non-structural subtype entailment with the following type signature:

$$\{\bot, \top, f\}$$

with three elements, where f is a single non-constant type constructor. With this limited signature, the non-structural subtype entailment is already PSPACE-hard [HR98], even without explicit reference to the constants $\bot$ and $\top$ in the constraints. With a reduction of the entailment problem to the universality problem of finite automata, which is PSPACE-complete [HU79], it is shown that a restricted version of the entailment on this simple signature is in fact PSPACE-complete [NP99].

Theorem 2.5.5 (Niehren & Priesnitz [NP99]) Under the signature $\{\bot, \top, f\}$, non-structural subtype entailment (to decide $C \models \alpha \leq \beta$) is PSPACE-complete if $\bot$ and $\top$ do not appear *explicitly* in the constraints.

If $\bot$ and $\top$ appear in the constraints, then the decidability of subtype entailment is still open, even for this restricted type signature. Niehren and Priesnitz recently give an equivalent characterization of entailment under the signature $\{\bot, \top, f\}$ [NP01]. This work extends the idea in [NP99] to reduce the entailment problem to an equivalent problem on an extended form of word automata, the so-called P-automata. However, the universality problem for P-automata appears to be an equally difficult problem, whose decidability is also open.

Theorem 2.5.6 (Niehren & Priesnitz [NP01]) Non-structural subtype entailment under the signature $\{\bot, \top, f\}$ is polynomially equivalent to the university problem of restricted P-automata.

Chapter 3

The First-order Theory

In this Chapter, we present the first decidability and undecidability results for the first-order theory of subtyping. We begin with introducing and motivating the study of the first-order theory of subtyping. The material in this chapter and the next chapter (Chapter 4) is an extended version of [SAN$^+$02].

3.1 First-order theory of subtyping constraints

Earlier, we have mentioned that constraint simplification is an important problem for designing scalable program analysis and type systems, and motivated the study of the constraint entailment problems. Despite extensive effort over many years, the exact complexity and even the decidability of entailment is open for some forms of subtyping constraints [Reh98,HR97,HR98,FF97,NP99,NP01,TS96,AWP97,FA96, Pot96,MW97,Pot01]. As we will see, the natural versions of entailment and subtyping constrained types can be encoded easily in the first-order theory of subtyping, so to gain insight into and take a step towards resolving these difficult problems, we study the full first-order theory.

We consider the following type language:

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

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

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

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

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

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

We next show how those entailment problems discussed in Chapters 1 and 2 fit in the first-order theory of subtyping.

To avoid confusion, recall that $\exists E$ is used as a shorthand for $\exists x_1, \ldots, \exists x_n$, where $E = \{x_1, \ldots, x_n\}$.

3.1.1 Entailment is in the $\forall$-fragment

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

The entailment problem $C \models x \leq y$ is in the universal fragment. Notice that C is a conjunction of basic constraints and the entailment $C \models x \leq y$ holds iff the universal formula $\forall x_1, \ldots, x_n.(C \rightarrow (x \leq y))$ is valid, where the x_i's are the variables free in $C \cup \{x \leq y\}$.

3.1.2 Existential entailment is in the $\forall\exists$-fragment

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

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

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

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

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

3.1.3 Subtyping constrained types is in the $\forall\exists$-fragment

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

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

where the α_i's and β_j's are the variables free in C_1 and C_2 respectively. Because C_1 and C_2 have disjoint sets of variables (see the definition of constrained types in Section 2.4), this is equivalent to:

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

In fact, we can show the following:

Proposition 3.1.1 Subtyping constrained types is polynomially reducible to existential entailment.

Proof. We have the following equivalences:

$$\alpha\backslash C_1 \leq \beta\backslash C_2$$

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

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

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

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

□

3.2 Undecidability of the first-order theory

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

The proof is via a reduction from the Post's Correspondence Problem (PCP) [Pos46] to a first-order formula of non-structural subtyping. Since PCP is undecidable [Pos46], the first-order theory of non-structural subtyping is undecidable as well. The proof follows the framework of Treinen [Tre92] and is inspired by the proof of undecidability of the first-order theory of ordering constraints over feature trees [MNT01].

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

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

where words are concatenated.

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

For the rest of this chapter, we consider the simple expression language:

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

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

3.2.1 Representing words as trees

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

Words as f-spines

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

Definition 3.2.1 (f-spine) A finite tree t (in f and $\perp$) is an f-spine if there is *exactly one maximal path* with labels f. On this maximal path, a left child represents 1 and a right child represents 2.

Example 2 (The word ϵ) The empty word ϵ is represented by the term $f(\perp, \perp)$. See Figure 3.1a.

Example 3 (The word 1) The word 1 is represented by the term $f(f(\perp, \perp), \perp)$. See Figure 3.1b.

Example 4 (The word 21221) The word 21221 is represented by the term:

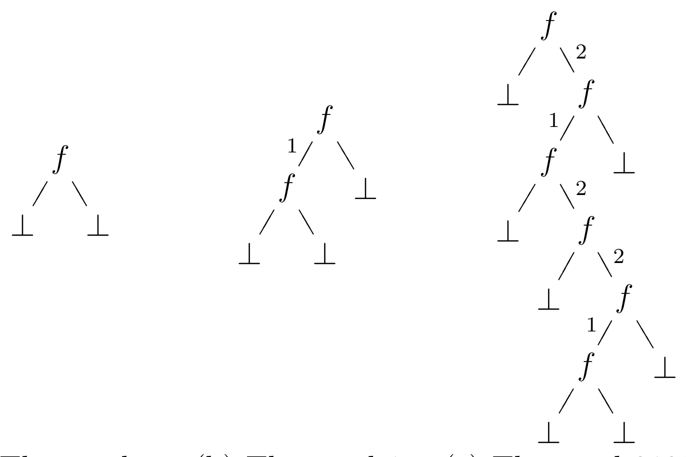
$$f(\perp, f(f(\perp, f(\perp, f(f(\perp, \perp), \perp))), \perp))$$

See Figure 3.1c.

Enforcing a word tree

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

1. Only f and $\perp$ appear in t (Lemma 3.2.2).



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

Figure 3.1: Some example representations of words.

2. There is exactly one maximal path of f's (Lemma 3.2.3).

3. t is not $\perp$ (because $\perp$ does not represent a word).

Lemma 3.2.2 A tree t contains only f and $\perp$ iff

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

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

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

Lemma 3.2.3 For any non-$\bot$ tree t with f and $\bot$, there is exactly one maximal path of f's iff the subtypes of t form a chain w.r.t. $\leq$.

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

Thus we can enforce a tree to represent a word. We use the shorthand $\mathsf{word}(t)$ for such a formula, which is constructed as follows:

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

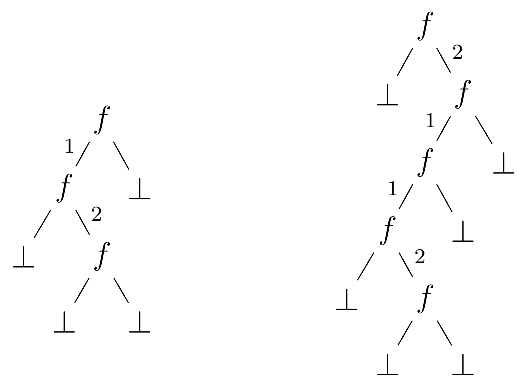
Prepending trees

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

To construct a solution to a PCP instance, we need to concatenate words. Thus we want to express with constraints that a word w_1 is obtained from w_2 by prepending w. We express this with a family of predicates $\mathsf{prepend}_w$, one for each constant word w. The predicate $\mathsf{prepend}_w(t_1, t_2)$ is true if the word represented by t_1 is obtained by prepending w to the word for t_2. Note that this is sufficient, because in PCP, the words are given as part of the problem. We define the predicate recursively:

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

Example 5 (Prepending example) We prepend the word 21 onto the word 12 (Figure 3.2a) to get the word 2112 (Figure 3.2b).



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

Figure 3.2: Tree prepending example.

3.2.2 Reducing PCP to FOT of subtyping

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

Outline of the reduction

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

We first describe a solution to a PCP instance as a tree. Recall that a PCP instance P consists of n pairs of words $\langle l_1, r_1\rangle, \ldots, \langle l_n, r_n\rangle$, where $l_i, r_i \in \{1, 2\}^*$. A solution $s = s_1 \cdots s_m$ to P is a *non-empty* finite sequence of indices 1 through n, *i.e.*, $s \in \{1, \cdots, n\}^+$, such that $l_{s_1} \cdots l_{s_m} = r_{s_1} \cdots r_{s_m}$. One can represent a solution s as the tree t shown in Figure 3.3. In the tree t, the values of ϵ, l_{s_1}, r_{s_1}, $\ldots$, $l_{s_m} \cdots l_{s_1}$, and $r_{s_m} \cdots r_{s_1}$ are represented by their corresponding word trees. The tree is constructed as follows. We start with the empty word pair $\langle \epsilon, \epsilon\rangle$. At each step, we prepend a particular pair from the PCP instance $\langle l_{s_i}, r_{s_i}\rangle$ to the previous pair of words. At the end, $l_{s_1} \cdots l_{s_m} = r_{s_1} \cdots r_{s_m}$, *i.e.*, we have found a solution to P. Notice that the solutions are constructed in the reverse order because we use prepend instead

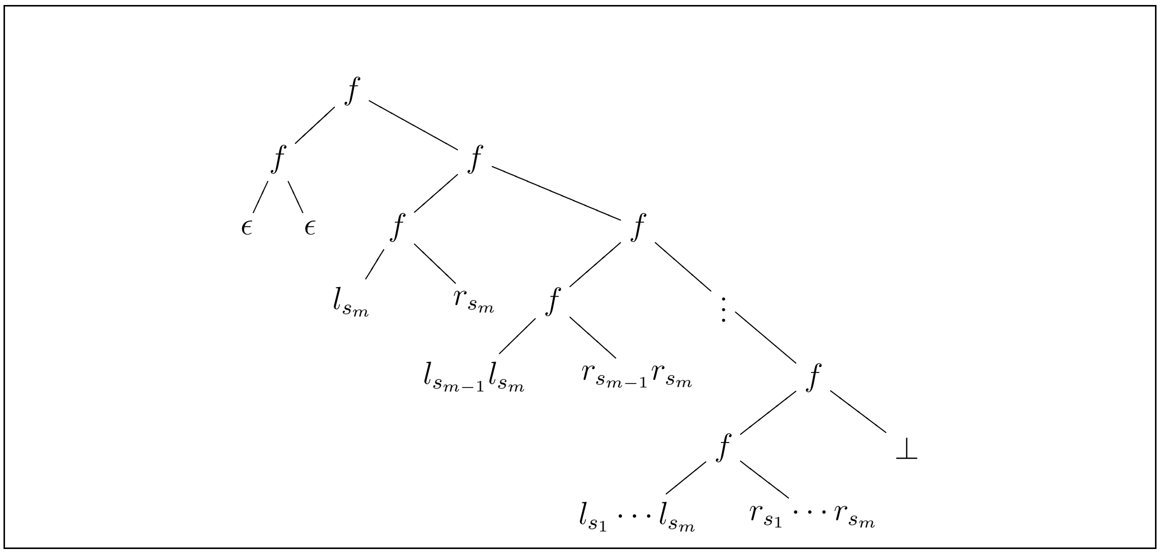


Figure 3.3: A PCP solution viewed as a tree.

of append. [1]

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

1. *The tree t is of the particular form in Figure 3.3.* (Section 3.2.2)

   Our construction does not require the *left* branches $f(w_i, w'_i)$ of the solution tree to be in the order shown in Figure 3.3. Any order is fine.

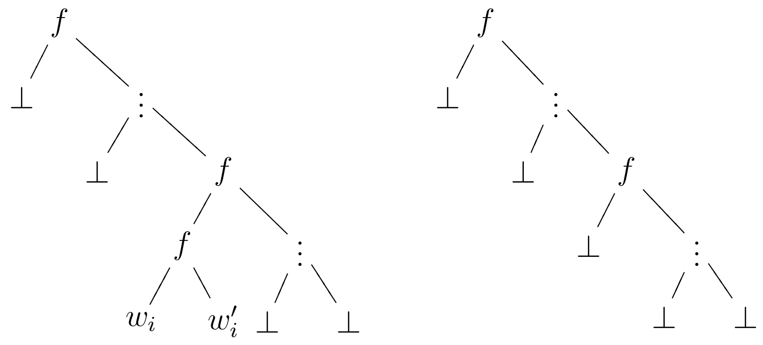
2. *We have a valid PCP construction sequence.* (Section 3.2.2)

   Each left branch $f(w_i, w'_i)$ is either the pair of empty words or there exists another left branch $f(w_j, w'_j)$ such that $\mathsf{prepend}_{l_k}(w_i, w_j)$ and $\mathsf{prepend}_{r_k}(w'_i, w'_j)$ for some k. In addition, one of the left branches is of the form $f(w, w)$ with w non-empty.[2] This ensures that we have a non-empty sequence.

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

[1]We use prepend because append is just not as convenient to express.

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



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

Figure 3.4: The left branch of a solution tree.

Correct form of the tree

To ensure the correct form of the tree t, we require that each left branch represents two words conjoined with the root labeled with f, *i.e.*, we have $f(w, w')$ for some trees representing words w and w'. In order to achieve this, we construct trees of the form shown in Figure 3.4a, which is a branch of the tree representing a PCP solution shown in Figure 3.3.

Let t be the tree representing a PCP solution. We cannot extract a left branch directly from t because subtyping constraints cannot express removing something from a tree. However, we observe that a left branch is a supertype of the *main spine* shown in Figure 3.4b with some additional properties, which we enforce separately. We first express the main spine s of t. Two properties are needed for s:

1. *The main spine s is of the form shown in Figure 3.4b.*

   We simply require $s \leq f(\perp, s)$.

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

   This is easily expressed as

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

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

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

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

1. *Exactly one left branch of the main spine is of the form* $f(w_i, w_i')$.

2. *All the other left branches of the main spine are labeled with* $\bot$.

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

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

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

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

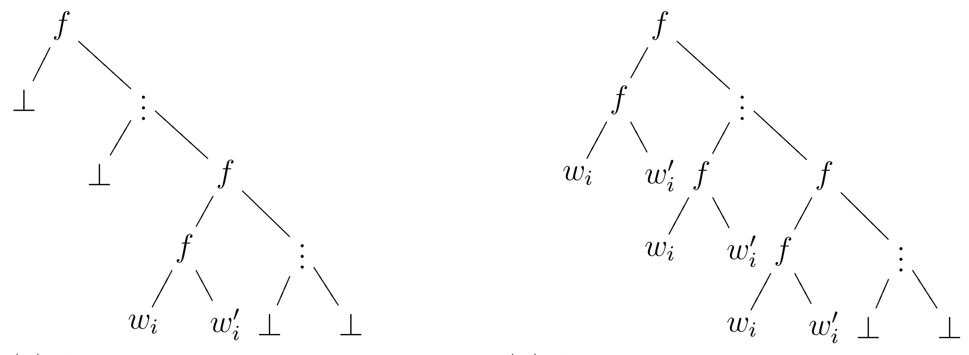
In addition, $\mathsf{uniq\text{-}min}(u, w)$ expresses that all the subtypes of u that are proper supertypes w have a *unique* minimal element, that is

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

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

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

We establish the correctness of $\mathsf{branch}(b, t)$ in Lemma 3.2.4.



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

Figure 3.5: Extracting words from a left branch.

Lemma 3.2.4 A tree b is a branch of t as shown in Figure 3.4a iff $\mathsf{branch}(b,t)$.

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

Correct construction of the tree

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

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

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

We give an example. Consider the left branch b shown in Figure 3.5a. We would like to build from b the expanded tree b' shown in Figure 3.5b. If we can construct such a tree b', then it is easy to extract the two words w_i and w_i' simply by the constraint $\exists u.f(f(w_i, w_i'), u) = b'$.

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

$$\mathsf{recurse}(t_1, t_2) \overset{\text{def}}{=} \begin{array}{l} (t_1 \leq t_2) \wedge \\ \exists x_1, x_2.(t_2 = f(x_1, x_2)) \ \wedge \ (x_2 \leq t_2) \end{array}$$

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

$$\mathsf{dup\text{-}branch}(b, b') \overset{\text{def}}{=} \begin{array}{l} \mathsf{recurse}(b, b') \wedge \\ \forall t.(\mathsf{recurse}(b, t) \ \rightarrow \ (b' \leq t)) \end{array}$$

We establish the correctness of $\mathsf{dup\text{-}branch}(b, b')$ in Lemma 3.2.5.

Lemma 3.2.5 Let b be a branch of t. A tree b' duplicates the non-$\perp$ sub-branch of b (as shown in Figure 3.5) iff $\mathsf{dup\text{-}branch}(b, b')$.

Proof. It is straightforward to verify that if b' duplicates the non-$\perp$ sub-branch of b, then $\mathsf{dup\text{-}branch}(b, b')$. For the other direction, assume $\mathsf{dup\text{-}branch}(b, b'')$. Since b' (shown in Figure 3.5b) meets the condition $\mathsf{recurse}(b, b')$, by definition of $\mathsf{dup\text{-}branch}$ we have $b'' \leq b'$. We also have $b \leq b''$ because $\mathsf{recurse}(b, b'')$ holds. With a simple induction on the height of the left spine of f's of b, we can show that b'' must be the same as b'. Thus, b'' duplicates the non-$\perp$ sub-branch of b. $\square$

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

$$\mathsf{wordpair}(w_1, w_2, b, t) \overset{\text{def}}{=} \begin{array}{l} \mathsf{word}(w_1) \ \wedge \ \mathsf{word}(w_2) \wedge \\ \exists b'.(\mathsf{dup\text{-}branch}(b, b') \wedge \\ \qquad \exists u.(f(f(w_1, w_2), u) = b')) \end{array}$$

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

$$\mathsf{onestep}(w_i, w'_i, w_j, w'_j) \stackrel{\text{def}}{=} \begin{array}{l} \bigvee_{1\le k\le n}(\mathsf{prepend}_{l_k}(w_i, w_j) \\ \qquad\qquad \wedge\ \mathsf{prepend}_{r_k}(w'_i, w'_j)) \end{array}$$

We can now express that the tree t represents a solution of a PCP instance. Recall that we must express that for each w_i and w'_i, either w_i and w'_i are the empty words, or there exist w_j and w'_j such that $\mathsf{prepend}_{l_k}(w_i, w_j)$ and $\mathsf{prepend}_{r_k}(w'_i, w'_j)$. Consider the PCP instance P in which we have $\langle l_1, r_1\rangle, \ldots, \langle l_n, r_n\rangle$, where l_i and r_i are words in $\{1, 2\}$. We construct a first-order formula $\mathsf{solvable}(P)$ which is valid iff P is solvable. The formula expresses the existence of a tree representing a solution to P.

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

$$\mathsf{empty}(w) \stackrel{\text{def}}{=} w = f(\bot, \bot)$$

$$\mathsf{construct}(w_1, w_2, b', t) \stackrel{\text{def}}{=} \begin{array}{l} \mathsf{branch}(b', t)\ \wedge \\ \exists w'_1, w'_2.(\mathsf{wordpair}(w'_1, w'_2, b', t) \\ \qquad \wedge\ \mathsf{onestep}(w_1, w_2, w'_1, w'_2)) \end{array}$$

$$\mathsf{valid\text{-}branch}(b, t) \stackrel{\text{def}}{=} \begin{array}{l} (\exists w_1, w_2.\mathsf{wordpair}(w_1, w_2, b, t) \\ \ \wedge\ ((\mathsf{empty}(w_1)\ \wedge\ \mathsf{empty}(w_2)) \\ \quad \vee\, \exists b'.\mathsf{construct}(w_1, w_2, b', t))) \end{array}$$

$$\mathsf{accept\text{-}branch}(b,t) \overset{\text{def}}{=} \begin{array}{l}\mathsf{branch}(b,t) \wedge \\ \exists w.(\mathsf{wordpair}(w,w,b,t) \wedge \\ \qquad \neg\mathsf{empty}(w))\end{array}$$

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

$$\mathsf{solvable}(P) \overset{\text{def}}{=} \begin{array}{l}\exists t.(\forall b.(\mathsf{branch}(b,t) \rightarrow \mathsf{valid\text{-}branch}(b,t)) \\ \qquad \wedge \exists b.\mathsf{accept\text{-}branch}(b,t))\end{array}$$

The correctness of the reduction from PCP to the first-order theory of subtyping constraints is established in Theorem 3.2.6.

Theorem 3.2.6 (Soundness and completeness) A PCP instance P has a solution iff the formula $\mathsf{solvable}(P)$ is valid.

Proof. It is easy to verify that if P has a solution, then any representation of the solution sequence in terms of a tree t shown in Figure 3.3 meets the requirement

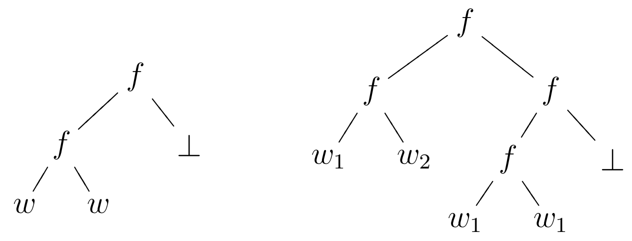
$$\forall b.(\mathsf{branch}(b,t) \rightarrow \mathsf{valid\text{-}branch}(b,t)) \wedge \exists b.\mathsf{accept\text{-}branch}(b,t)$$

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

3.2.3 Recursive types

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

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



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

Figure 3.6: Failed attempts for recursive types.

It turns out that only the words we get from a left branch of t must be finite. The other trees in the construction can be infinite. For words, if we do not restrict them to be finite, the existence of such a tree t as in Figure 3.3 may not correspond to a solution to the PCP problem. To see this, consider the PCP instance $\{\langle 11, 1\rangle\}$. Clearly, it has no solution. However, consider the tree $(f(f(w, w), \bot)$ shown in Figure 3.6a, where w is the infinite regular tree such that $w = f(w, \bot)$, *i.e.*, the infinite word 1^ω.

One may wonder whether we can instead require that a construction step must use two different branches, and that the words for the two branches are not the same at the respective positions. This does not work either. Consider the PCP instance $\{\langle \epsilon, 1\rangle, \langle \epsilon, 2\rangle\}$, which has no solution. Now consider the tree $f(f(w_1, w_2), f(f(w_1, w_1), \bot))$ shown in Figure 3.6b, where $w_1 = f(w_2, \bot) \ \wedge \ w_2 = f(\bot, w_1)$, *i.e.*, w_1 is the infinite word $(12)^\omega$ and w_2 is the infinite word $(21)^\omega$.

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

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

Lemma 3.2.7 A tree t representing a word is finite iff $\mathsf{has\text{-}max}(t)$.

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

We can now directly use the construction in Section 3.2, except we require in the formula $\mathsf{wordpair}(w_1, w_2, b, t)$ that w_1 and w_2 are finite:

$$\mathsf{wordpair}(w_1, w_2, b, t) \overset{\text{def}}{=} \begin{array}{l} \mathsf{word}(w_1) \ \wedge \ \mathsf{word}(w_2) \wedge \\ \mathsf{has\text{-}max}(w_1) \ \wedge \ \mathsf{has\text{-}max}(w_2) \wedge \\ \exists b'.(\mathsf{dup\text{-}branch}(b, b') \wedge \\ \quad \exists u.(f(f(w_1, w_2), u) = b')) \end{array}$$

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

Theorem 3.2.8 The first-order theory of non-structural subtyping constraints over recursive types (and infinite trees) is undecidable for any type language with a binary type symbol and $\perp$.

Proof. Follows from Lemma 3.2.7 and Theorem 3.2.6. $\square$

3.3 Structural subtyping: a comparison

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

Theorem 3.3.1 The first-order theory of structural subtyping constraints with a single constant symbol is decidable for both simple and recursive types (and infinite trees).

Proof. This can be easily shown by noticing that in a type language with only one constant (*i.e.*, $\perp$), the subtype relation is the same as equality. Thus we can simply turn any constraint $t_1 \leq t_2$ into $t_1 = t_2$. Since the first-order theory of equality is decidable both for finite and regular trees (and infinite trees) [Mah88], the theorem follows immediately. □

It is open whether the first-order theory of structural subtyping constraints is decidable in general, where arbitrary base lattices are allowed. We suspect a quantifier elimination procedure similar to that of [Mah88] for the first-order theory of equality can be constructed, which we leave as interesting future research. If the first-order theory of structural subtyping constraints turns out to be decidable, then we have obtained another level of separation of structural and non-structural subtyping.

Chapter 4

Tree Automata and Entailment

In the last chapter (Chapter 3), we have presented some negative results on constraint simplification and entailment. In this chapter and Chapter 5, we present positive results for some special cases. In particular, in this chapter, we introduce an approach based on tree automata for subtype entailment, and we show how to apply this technique to prove that the monadic fragment of the first-order theory of subtyping is decidable. We begin this chapter by introducing some necessary background on tree automata [CDG+99, GS84].

4.1 Background on tree automata

We recall some definitions and results on tree automata [CDG+99, GS84].

Tree automata generalize word automata by accepting trees instead of words. Let Σ be a ranked alphabet, and let Σ_n denote the set of symbols of arity n. We recall the definition of trees from Chapter 2.

Definition 4.1.1 (Finite tree) A *finite tree* t over a ranked alphabet Σ is a mapping from a prefix-closed set $\text{pos}(t) \subseteq \mathbb{N}^*$ into Σ. The set of *positions* pos of t satisfies:

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

Definition 4.1.2 (Finite tree automata (NFTA)) A *finite tree automaton* (NFTA) over Σ is a tuple

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

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

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

where $n \geq 0$, $f \in \Sigma_n$, $q, q_1, \ldots, q_n \in Q$.

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

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

Definition 4.1.3 (Run) A *run* $\mathbb{R}$ of an automaton $\mathcal{A} = (Q, \Sigma, Q_F, \Delta)$ on a term t is a labeling of pos(t) with states Q of $\mathcal{A}$ such that for every position $\pi \in \text{pos}(t)$, if $t(\pi) = f \in F_n$, $\mathbb{R}(\pi) = q$, and $\mathbb{R}(\pi i) = q_i$ for each $1 \leq i \leq n$, then $f(q_1, \cdots, q_n) \longrightarrow q \in \Delta$. A run is *successful* if $\mathbb{R}(\epsilon) \in Q_F$.

Example 6 (tree automaton) Consider the automaton where

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

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

$$\begin{array}{ccccccc} f & \longrightarrow & f & \longrightarrow & f & \longrightarrow & q_f \\ / \ \backslash & & / \ \backslash & & / \ \backslash & & \\ a \quad b & & q \quad b & & q \quad q_f & & \end{array}$$

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

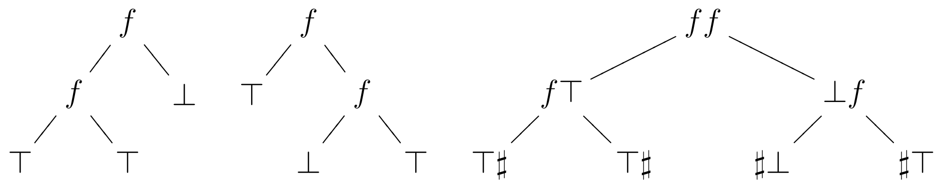
We use a standard encoding to represent tuples [CDG+99]. We first give an example to illustrate how the encoding works. Consider tuples of words over the alphabet $\{0, 1\}$. We can construct an automaton to accept the (encoding of a) language L of pairs (w, w') such that $\|w\| = \|w'\|$ (where $\|w\|$ denotes the length of the word w) and $w_i \neq w'_i$ for $1 \leq i \leq \|w\|$, *i.e.*, we flip 0's and 1's in w and w'. One possible encoding is to "stack" the two words, *i.e.*, put one on top of the other, and we consider the product alphabet $\{\genfrac{}{}{0pt}{}{0}{0}, \genfrac{}{}{0pt}{}{0}{1}, \genfrac{}{}{0pt}{}{1}{0}, \genfrac{}{}{0pt}{}{1}{1}\}$. With this encoding, we can easily construct an automaton that accepts L, for example, the automaton with one state q and q is both initial and final, having transitions $(q, \genfrac{}{}{0pt}{}{0}{1}) \longrightarrow q$ and $(q, \genfrac{}{}{0pt}{}{1}{0}) \longrightarrow q$.

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

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

$\{af, fa, ff, f\sharp, \sharp f\}$.

Example 7 (tuple encoding) This example shows how to encode the tuple (t_1, t_2):



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

Definition 4.1.4 (Tree automata on tuples) Let Σ be a ranked alphabet. A *finite tree automaton on n-tuples* over Σ is a tree automaton $\mathcal{A} = (Q, \Sigma^n, Q_F, \Delta)$ over Σ^n (defined above), where Q is a finite set of *states*, $Q_F \subseteq Q$ is a set of *final states*, and Δ is a set of *transition rules* of the form

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

where $n \geq 0$, $f \in \Sigma^n_m$, $q, q_1, \ldots, q_m \in Q$.

Example 8 (automaton on tuples) Consider the automaton where

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

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

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

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

Definition 4.1.5 (Projection and cylindrification) If $R \subseteq T(\Sigma)^n$ $(n \geq 1)$ and $1 \leq i \leq n$, then the *i-th projection* of R is the relation $R^i \subseteq T(\Sigma)^{n-1}$ defined by

$$R^i(t_1, \ldots, t_{n-1}) \;\Leftrightarrow\; \exists t \in T(\Sigma).R(t_1, \ldots, t_{i-1}, t, t_i, \ldots, t_{n-1})$$

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

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

We summarize here results on tree automata that we use. More details can be found in [GS84, CDG+99].

Definition 4.1.6 (Tree automata emptiness) *The tree automata emptiness problem is that given any tree automata to decide whether it accepts any trees.*

Theorem 4.1.7 (Decidable emptiness) The emptiness problem for tree automata is decidable. In fact, it can be decided in linear time in the size of the automaton.

Theorem 4.1.8 (Closure properties) Tree automata are closed under intersection, union, complementation, cylindrification, and projection.

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

4.2 Decidability of the monadic fragment

In this section, we show that the monadic fragment of the first-order theory (*i.e.*, the fragment with only unary function symbols and constants) is decidable. This result shows that the difficulty in the whole first-order theory and in various entailment problems lies in binary type constructors. The idea of the proof is to reduce the problem to the tree automata emptiness problem, or equivalently, to WS1S or S1S [Tho96, Tho90].

Note that word automata would suffice for encoding the case with unary function symbols. However, because our approach is extensible to type languages over arbitrary signatures for the existential or universal fragments (see Section 4.3), we present our results in terms of tree automata.

Recall that we consider a monadic signature in this section. We reduce the validity of a formula ϕ to the emptiness decision of a tree automaton. We proceed by structural induction on the formula ϕ. We assume the formula is normalized so that it uses only the connectives $\wedge$, $\neg$, and $\exists$. In addition, w.l.o.g., we assume the literals of the formula are of the form $x \leq y$, $x = \bot$, $x = \top$, and $x = f(y)$, which can be obtained by the following standard flattening procedure. There is a related definition of *one-level systems* in solving set constraints [AW92]. We show that a constraint can be translated to flat constraints in linear time and generate flat constraints linear in size in the size of the original constraint. The procedure is easily extended to work on quantified formulas by introducing appropriate quantifiers for the temporary variables created. We present the procedure in terms of a binary type constructor $\times$.

Definition 4.2.1 (Flat constraints) A constraint is *flat* if it is of one of the following forms

- $\alpha \leq \beta$;
- $\alpha = \bot$;
- $\alpha = \top$;
- $\alpha = \beta \times \gamma$

where α, β, and γ are variables.

Proposition 4.2.2 For any constraint $c = \tau_1 \leq \tau_2$, there exist equivalent flat constraints C, and the flat constraints C can be generated in linear time and are of linear size in c.

Proof. Perform the following transformation FLATTEN on types

- FLATTEN$(\bot) = (\alpha, \{\alpha = \bot\})$, where α is fresh;
- FLATTEN$(\top) = (\alpha, \{\alpha = \top\})$, where α is fresh;
- FLATTEN$(\alpha) = (\alpha, \{\})$;
- FLATTEN$(\tau_1 \times \tau_2) = (\alpha, \{\alpha = \alpha_1 \times \alpha_2\} \cup C_1 \cup C_2)$, where $(\alpha_1, C_1) =$ FLATTEN(τ_1) and $(\alpha_2, C_2) =$ FLATTEN(τ_2), and α is a fresh variable.

For a constraint $\tau_1 \leq \tau_2$, let $(\alpha_1, C_1) =$ FLATTEN(τ_1) and $(\alpha_2, C_2) =$ FLATTEN(τ_2), we construct the flat constraint system $C_1 \cup C_2 \cup \{\alpha_1 \leq \alpha_2\}$. □

We now give our construction, which proceeds according to the structure of the given formula.

- $\exists x.\phi$

  Let $\mathcal{A}_1$ be the automaton for ϕ. We construct an automaton $\mathcal{A}$ for $\exists x.\phi$ by taking the projection of $\mathcal{A}_1$ w.r.t. the x component of the tuple.[1]

- $\neg\phi$

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

- $\phi_1 \wedge \phi_2$

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

The following are for the base predicates.

- $x = \bot$

  We construct the automaton

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

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

- $x = \top$

  We construct the automaton

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

- $x = f(y)$

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

  We construct the following automaton

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

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

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

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

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

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

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

Here are the base cases.

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

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

- $x \leq y$

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

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

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

  We construct the automaton

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

  The transition relation Δ is constructed in pieces.

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

  $$\begin{array}{rcl} \bot\bot & \longrightarrow & q_f \\ \bot\top & \longrightarrow & q_f \\ \top\top & \longrightarrow & q_f \end{array}$$

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

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

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

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

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

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

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

$$ff(q_f) \longrightarrow q_f$$

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

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

Theorem 4.2.3 The first-order theory of non-structural subtyping with unary function symbols is decidable. This holds both for the finite and infinite words and for structural subtyping as well.

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

4.3 Extending to arbitrary signatures

We now discuss the issues with extending the described approach to arbitrary signatures. There are two related difficulties in extending our approach to the full first-order theory over arbitrary signatures. First, although we can easily express the solutions to $x \leq y$ with standard tree automata, we cannot express the solutions to $x = f(y, z)$ with standard tree automata for any binary symbol f, because the set $\{\langle t_1, t_2, t_3 \rangle \mid t_1 = f(t_2, t_3)\}$ is not a regular set [CDG+99]. An extended form of tree automata on tuples is required, *tree automata on tuples with component-wise tests* (TACT); such automata allow machines to test relationships between tuple components [Tre00]. Because this class of tree automata is not closed under projection, an operation needed for existential quantifier elimination, it does not extend to the full first-order theory. However, this class of automata is still interesting because it can encode the existential or equivalently the universal fragments of the first-order theory. Therefore, we can reduce non-structural subtype entailment to the emptiness problem on a restricted class of TACT. We believe this reduction is a promising direction in resolving the decidability of non-structural subtype entailment.

4.3.1 Expressing $x \leq y$ in general

We first show how to encode $x \leq y$ in the general case (with a binary constructor f). For $x \leq y$ to hold, we have the following cases:

- x is $\perp$;
- y is $\top$;
- $x = f(x_1, x_2)$ and $y = f(y_1, y_2)$, where $x_1 \leq y_1$ and $x_2 \leq y_2$.

We construct the automaton

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

We construct the transition relation Δ in pieces.

We have the atomic cases where x and y are either $\bot$ or $\top$:

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

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

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

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

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

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

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

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

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

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

For an additional example of a direct construction, we give the construction for $x \not\leq y$, although it can also be obtained by complementing the automaton constructed for $x \leq y$. The construction of this automaton is similar to the one for $x \leq y$. The reader is invited to verify the construction.

We construct the automaton

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

where Δ is given by

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

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

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

$$\begin{array}{rcl}
\bot\bot & \longrightarrow & q \\
\bot\top & \longrightarrow & q \\
\bot f(q_l, q_l) & \longrightarrow & q \\
ff(q, q) & \longrightarrow & q \\
f\bot(q_r, q_r) & \longrightarrow & q \\
f\top(q_r, q_r) & \longrightarrow & q \\
\top\bot & \longrightarrow & q \\
\top f(q_l, q_l) & \longrightarrow & q \\
\top\top & \longrightarrow & q
\end{array}$$

The meaning of states q_l and q_r are the same as in the case $x \leq y$. The state q is used to recognize all the possible pairs of terms. The constructions above can also be easily extended to handle type constructors with contravariant fields.

4.3.2 Constrained automata for $x = f(y, z)$

The only literals that cannot be expressed with a standard tree automaton are of the form $x = f(y, z)$. The idea is to separate the regular part of constraints (those that can be expressed with tree automata) and the non-regular part (which are not expressible as tree automata).

We first introduce a notion of a constrained tree automaton, which is a special case of tree automata with component-wise tests [Tre00], such that the tests are only performed at the root of the tree being accepted. Interested readers are referred to [Tre00, CDG+99] for more details on tree automata with tests.

Consider a tree automaton $\mathcal{A}$ on n-tuples over the ranked alphabet Σ. We name its n components $x_1, \ldots, x_n$. We are interested in the following decision problem: Given $\mathcal{A}$ and C, where C is a set of equations in terms of $x_1, \ldots, x_n$ and the alphabet Σ (these are unification constraints), we want to decide whether there exist trees

$t_1, \ldots, t_n$ such that:

1. $\langle t_1, \ldots, t_n \rangle$ is accepted by $\mathcal{A}$.

2. The valuation h with $h(x_i) = t_i$ satisfies C.

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

Example 9 (constrained automata) Consider the automaton $\mathcal{A}$ where

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

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

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

4.3.3 Expressing subtype entailment

We now show subtype entailment (structural or non-structural, finite or recursive) can be reduced to the emptiness problem for constrained automata that we have just introduced.

Consider the entailment problem $C \vDash x \leq y$. It is easy to see that the entailment holds if and only if the constraint $C' = C \wedge x \not\leq y$ does not have a solution. Each solution to C' corresponds to a witness to the non-entailment of $C \vDash x \leq y$. The idea is to use a constrained automaton to express all the solutions to the constraint $C \wedge x \not\leq y$. Let $C = \{\tau_1 \leq \tau_1', \ldots, \tau_n \leq \tau_n'\}$. Then C is equivalent to the single constraint

$$\tau \leq \tau'$$

where $\tau = f(\tau_1, f(\tau_2, f(\dots, \tau_n)))$ and $\tau' = f(\tau'_1, f(\tau'_2, f(\dots, \tau'_n)))$. For example, let C be the following set of constraints:

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

It is equivalent to the constraint:

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

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

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

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

This is a very simple and straightforward reduction. There are some special properties about the constructed automaton for entailment. The tree automata part consists of an automaton with a bounded number of states. Next, we show that general constrained automata emptiness is undecidable. The proof crucially relies on that the associated tree automaton has unbounded number of states. Therefore, it is open whether constrained automata emptiness is decidable if the associated tree automaton has a bounded number of states.

4.3.4 Constrained automata emptiness is undecidable

The problem of deciding whether a constrained automaton accepts the empty language is undecidable. This holds for the smallest signature $f(,)$ and $\bot$. This is the smallest because we can show that if we only have unary function symbols and constants, emptiness is decidable (the proof is similar to the proof for the monadic fragment). The proof of undecidability is through a reduction from PCP to the emptiness problem of constrained automata.

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

We adopt and simplify a proof of Treinen [Tre01] for constrained automata. We first recall the encoding of words over 0 and 1 with trees over $f(,)$ and $\perp$:

- $c(\epsilon) = f(\perp, \perp)$;
- $c(0w) = f(c(w), \perp)$;
- $c(1w) = f(\perp, c(w))$.

We construct an automaton $\mathcal{A}$ to accept all the encodings of the pairs of the words $\langle w, w', wp_i, w'q_i \rangle$ for some p_i and q_i, *i.e.*,

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

This automaton expresses a single step in the PCP construction. For a particular i, we construct an automaton to accept all the tuples $\langle w, wv \rangle$ (where $v = v_1 \dots v_n$) and then use the fact that tree automata are closed under cylindrification and union to construct $\mathcal{A}$ for the above defined tree language.

We have the following set of rules:

$$ff(q_f, q_\perp) \longrightarrow q_f \tag{4.1}$$
$$ff(q_\perp, q_f) \longrightarrow q_f \tag{4.2}$$
$$ff(q_1, q_\perp) \longrightarrow q_f \quad \text{if } v_1 = 0 \tag{4.3}$$
$$ff(q_\perp, q_1) \longrightarrow q_f \quad \text{if } v_1 = 1 \tag{4.4}$$
$$\perp f(q_i, q_{\sharp\perp}) \longrightarrow q_{i-1} \quad \text{if } v_i = 0 \tag{4.5}$$
$$\perp f(q_{\sharp\perp}, q_i) \longrightarrow q_{i-1} \quad \text{if } v_i = 1 \tag{4.6}$$
$$\perp\perp \longrightarrow q_\perp \tag{4.7}$$
$$\sharp\perp \longrightarrow q_{\sharp\perp} \tag{4.8}$$
$$\sharp\perp \longrightarrow q_n \tag{4.9}$$

For $n = 0$, we simply have

$$ff(q_f, q_\perp) \longrightarrow q_f \tag{4.10}$$
$$ff(q_\perp, q_f) \longrightarrow q_f \tag{4.11}$$
$$\perp\perp \longrightarrow q_f \tag{4.12}$$

With the above construction, we build an automaton for each word v (either p_i or q_i) in the PCP instance such that the automaton accepts the language $\langle w, wv\rangle$. With these automata as sub-automata, we can construct an automaton $\mathcal{A}$ to accept the language that represents a PCP step, *i.e.*, $\langle w, w', wp_i, w'q_i\rangle$ for some i.

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

$$\perp\perp \longrightarrow q_f \tag{4.13}$$

$$ff(q_f, q_f) \longrightarrow q_f \tag{4.14}$$

By applying cylindrification twice on this automaton, we get an automaton that accepts the language $\langle w_1, w_2, w, w\rangle$.

Taking the intersection of this automaton with the previously constructed automaton $\mathcal{A}$, we get $\mathcal{A}'$. We cylindrify both automata $\mathcal{A}$ and $\mathcal{A}'$ to add a fifth component. For simplicity, we reuse $\mathcal{A}$ and $\mathcal{A}'$ for the resulting automata. Notice that both $\mathcal{A}$ and $\mathcal{A}'$ have only a single accepting state, denoted by p_f and q_f respectively.

We construct the final automaton with the automata $\mathcal{A}$ and $\mathcal{A}'$ as its sub-automata. Here are the rules:

$$\perp\perp\perp\perp_{-} \longrightarrow p_\perp \tag{4.15}$$

$$\perp\perp ff_{-}(p_\perp, p_\perp) \longrightarrow p_0 \tag{4.16}$$

$$ffff_{-}(p_0, p_f) \longrightarrow p_0 \tag{4.17}$$

$$ffff_{-}(p_0, q_f) \longrightarrow p \tag{4.18}$$

where p is the only accepting state.

We denote the constructed automaton as $\mathcal{A}_P$, and name the five components of $\mathcal{A}_P$ with the variables x,y,u,v, and w. The associated constraints are $u = f(x, w)$ and $v = f(y, w)$. This completes our reduction from PCP to the constrained automata emptiness problem.

Theorem 4.3.1 (Soundness and completeness) The PCP instance P is solvable iff there is a tuple of trees $\langle x, y, u, v, w\rangle \in L(\mathcal{A}_P)$ and satisfies the unification constraints $u = f(x, w)$ and $v = f(y, w)$.

4.4 An example

We give an example in this section to demonstrate the automata construction of Section 4.2. Consider the alphabet $\Sigma = \{\bot, \top, g(\cdot)\}$. We want to decide the entailment $\{x \leq g(y), g(x) \leq y\} \models x \leq y$.

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

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

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

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

We construct an automaton for each of the five constraints.

- $x' = g(x)$

  Consider the automaton where

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

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

- $y' = g(y)$

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

- $x \leq y'$

  Consider the automaton where

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

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

- $x' \leq y$

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

- $x \not\leq y$

Consider the automaton where

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

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

Now we apply cylindrification to the automata above.[2] We use the following shorthand for transition rules:

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

is a shorthand for the two rules

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

and

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

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

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

$$
\begin{array}{rcl}
Q & = & \{q_1, q_2, q_f\} \\
\Sigma & = & \{\bot, \top, g(\cdot)\}^4 \\
Q_F & = & \{q_f\} \\
\Delta & = & \left\{ \begin{array}{ccc}
\multicolumn{3}{l}{\text{/* derived from } \sharp\bot \longrightarrow q_1 \text{ */}} \\
\sharp(\sharp \mid \bot \mid \top)\bot(\sharp \mid \bot \mid \top) & \longrightarrow & q_1 \\
\sharp g\bot(\sharp \mid \bot \mid \top \mid g)(q_1) & \longrightarrow & q_1 \\
\sharp g\bot(\sharp \mid \bot \mid \top \mid g)(q_2) & \longrightarrow & q_1 \\
\sharp g\bot(\sharp \mid \bot \mid \top \mid g)(q_f) & \longrightarrow & q_1 \\
\sharp(\sharp \mid \bot \mid \top)\bot g(q_1) & \longrightarrow & q_1 \\
\sharp(\sharp \mid \bot \mid \top)\bot g(q_2) & \longrightarrow & q_1 \\
\sharp(\sharp \mid \bot \mid \top)\bot g(q_f) & \longrightarrow & q_1 \\
 & & \\
\multicolumn{3}{l}{\text{/* derived from } \bot g(q_1) \longrightarrow q_f \text{ */}} \\
\bot g(q_1) & \longrightarrow & q_f \\
\bot(\sharp \mid \bot \mid \top \mid g)g(\sharp \mid \bot \mid \top \mid g)(q_1) & \longrightarrow & q_f \\
 & & \\
\multicolumn{3}{l}{\text{/* derived from } \sharp\top \longrightarrow q_2 \text{ */}} \\
\sharp(\sharp \mid \bot \mid \top)\top(\sharp \mid \bot \mid \top) & \longrightarrow & q_2 \\
\sharp g\top(\sharp \mid \bot \mid \top \mid g)(q_1) & \longrightarrow & q_2 \\
\sharp g\top(\sharp \mid \bot \mid \top \mid g)(q_2) & \longrightarrow & q_2 \\
\sharp g\top(\sharp \mid \bot \mid \top \mid g)(q_f) & \longrightarrow & q_2 \\
\sharp(\sharp \mid \bot \mid \top)\top g(q_1) & \longrightarrow & q_2 \\
\sharp(\sharp \mid \bot \mid \top)\top g(q_2) & \longrightarrow & q_2 \\
\sharp(\sharp \mid \bot \mid \top)\top g(q_f) & \longrightarrow & q_2 \\
 & & \\
\multicolumn{3}{l}{\text{/* derived from } \top g(q_2) \longrightarrow q_f \text{ */}} \\
\top(\sharp \mid \bot \mid \top \mid g)g(\sharp \mid \bot \mid \top \mid g)(q_2) & \longrightarrow & q_f \\
 & & \\
\multicolumn{3}{l}{\text{/* derived from } gg(q_f) \longrightarrow q_f \text{ */}} \\
g(\sharp \mid \bot \mid \top \mid g)g(\sharp \mid \bot \mid \top \mid g)(q_f) & \longrightarrow & q_f
\end{array} \right\}
\end{array}
$$

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

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

- $y' = g(y)$
- $x \leq y'$
- $x' \leq y$
- $x \nleq y$

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

Chapter 5

Conditional Equality Constraints

In this chapter, we consider the entailment problem for *conditional equality constraints*, a special form of non-structural subtyping constraints. The material in this chapter is an extended version of [SA01].

Conditional equality constraints extend the usual equality constraints with an additional form $\alpha \Rightarrow \tau$, which holds if $\alpha = \bot$ or $\alpha = \tau$. Conditional equality constraints have been used in a number of analyses, such as the tagging analysis of Henglein [Hen92], and the pointer analysis proposed by Steensgaard [Ste96], and a form of equality-based flow systems for higher order functional languages [Pal98].

Besides conditional equality constraints, we also consider entailment for a natural extension of conditional constraints (Section 5.5). In particular, we show there are polynomial time algorithms for equality and conditional equality constraints for both versions of entailment. We believe these algorithms may be of practical interest. In addition, we consider simple entailment and restricted entailment for a simple and natural extension of conditional equality constraints. We show that although simple entailment for the extension admits polynomial time algorithms, restricted entailment for this extension turns out to be coNP-complete. The coNP-completeness result is interesting because it provides one natural boundary between tractable and intractable constraint languages. The results are summarized in Table 5.1.

	Complexity	
Constraints	Simple Entailment	Restricted Entailment
Conditional Equality	P-complete	P-complete
Extended Conditional Equality	P-complete	coNP-complete

Table 5.1: Summary of results.

5.1 The problem setting

In this section, we recall some definitions from Chapter 2, specialized now to conditional equality constraints. We work with simple types. The algorithms we present apply to type languages with other base types and type constructors. Our type language is:

$$\tau ::= \bot \mid \top \mid \tau_1 \rightarrow \tau_2 \mid \alpha.$$

This simple language has two constants $\bot$ and $\top$, a binary constructor $\rightarrow$, and variables α ranging over a denumerable set $\mathcal{V}$ of type variables. $\mathcal{T}$ and $\mathcal{T}_{\mathcal{G}}$ are the set of types and the set of ground types respectively. An *equality constraint* is $\tau_1 = \tau_2$ and a *conditional equality constraint* is $\alpha \Rightarrow \tau$. A *constraint system* is a finite conjunction of equality and conditional equality constraints. An *equality constraint system* has only equality constraints.

A valuation ρ *satisfies* constraint $\tau_1 = \tau_2$, written $\rho \vDash \tau_1 = \tau_2$, if $\rho(\tau_1) = \rho(\tau_2)$, and it satisfies a constraint $\alpha \Rightarrow \tau$, written $\rho \vDash \alpha \Rightarrow \tau$, if $\rho(\alpha) = \bot$ or $\rho(\alpha) = \rho(\tau)$. We write $\rho \vDash C$ if ρ satisfies every constraint in C.

Definition 5.1.1 (Terms) Let C be a set of constraints. Term(C) is the set of terms appearing in C: $\text{Term}(C) = \{\tau_1, \tau_2 \mid (\tau_1 = \tau_2) \in C \vee (\tau_1 \Rightarrow \tau_2) \in C\}$.

The satisfiability of equality constraints can be decided in almost linear time in the size of the original constraints using a union-find data structure [Tar75]. With a simple modification to the algorithm for equality constraints, we can decide the satisfiability of a system of conditional equality constraints in almost linear time (see Proposition 5.1.2 below).

Example 10 Here are example conditional constraints:

- $\alpha \Rightarrow \bot$;

  Solution: α must be $\bot$

- $\alpha \Rightarrow \top$;

  Solution: α is either $\bot$ or $\top$.

- $\alpha \Rightarrow \beta \to \gamma$

  Solution: α is either $\bot$ or a function type $\beta \to \gamma$, where β and γ can be any type.

Proposition 5.1.2 Let C be any system of constraints with equality constraints and conditional equality constraints. We can decide whether there is a satisfying valuation for C in almost linear time.

Proof. The basic idea of the algorithm is to solve the equality constraints and to maintain along with each variable a list of constraints conditionally depending on that variable. Once a variable α is unified with a non-$\bot$ value, any constraints $\alpha \Rightarrow \tau$ on the list are no longer conditional and are added as equality constraints $\alpha = \tau$. The time complexity is still almost linear since each constraint is processed at most twice. See, for example, [Ste96] for more information. □

We refer to this algorithm as CondResolve. The result of running the algorithm on C is denoted by CondResolve(C).

In this chapter, we consider the usual two forms of entailment for conditional equality constraints: [1]

- *simple entailment*: $C \vDash c$, and
- *restricted entailment*: $C_1 \vDash_E C_2$

[1]To follow the convention used in the literature in program analysis, we use restricted entailment in stead of existential entailment in this section, although they are equivalent.

where C, C_1, and C_2 are systems of constraints, and c is a single constraint, and E is a set of *interface* variables.

For the use of restricted entailment, consider the following situation. In a polymorphic analysis, a function (or a module) is analyzed to generate a system of constraints [FFA00, FF97]. Only a few of the constraint variables are visible outside the function, call them the *interface variables.* We would like to simplify the constraints with respect to the set of interface variables. This can give us better simplification because we only need to preserve the solutions of a smaller set of variables. Thus, in practice, restricted entailment is more commonly encountered than simple entailment.

For a simpler presentation of the algorithms and a closer match with the program analysis literature, we define the two entailment problems slightly differently from the ones given in Chapter 2.

Definition 5.1.3 (Simple entailment) Let C be a system of constraints and c a constraint. We say that $C \vDash c$ if for every valuation ρ with $\rho \vDash C$, we have $\rho \vDash c$ also.

Definition 5.1.4 (Restricted entailment) Let C_1 and C_2 be two constraint systems, and let E be the set of variables $\mathrm{fv}(C_1) \cap \mathrm{fv}(C_2)$. We say that $C_1 \vDash_E C_2$ if for every valuation ρ_1 with $\rho_1 \vDash C_1$ there exists ρ_2 with $\rho_2 \vDash C_2$ and $\rho_1(\alpha) = \rho_2(\alpha)$ for all $\alpha \in E$.

Definition 5.1.5 (Interface and internal variables) In $C_1 \vDash_E C_2$, variables in E are *interface variables.* Variables in $(\mathrm{fv}(C_1) \cup \mathrm{fv}(C_2)) \setminus E$, are *internal variables.*

We adopt the following notational convention in this chapter:

- τ and τ_i denote type expressions.
- α, β, γ, α_i, β_i, and γ_i denote interface variables.
- μ, ν, σ, μ_i, ν_i, and σ_i denote internal variables.
- α denotes a generic variable, in places where we do not distinguish interface and internal variables.

For simple entailment $C \models c$, it suffices to consider only the case where c is a constraint between variables, *i.e.*, c is of the form $\alpha = \beta$ or $\alpha \Rightarrow \beta$. For simple entailment, $C \models \tau_1 = \tau_2$ if and only if $C \cup \{\alpha = \tau_1, \beta = \tau_2\} \models \alpha = \beta$, where α and β do not appear in C and c. The same also holds for when c is of the form $\alpha \Rightarrow \tau$.

Simple entailment also enjoys a distributive property, that is $C_1 \models C_2$ if and only if $C_1 \models c$ for each $c \in C_2$. Thus it suffices to study only $C \models c$. This distributive property does not hold for restricted entailment. Consider $\emptyset \models_{\{\alpha,\beta\}} \{\alpha \Rightarrow \sigma, \beta \Rightarrow \sigma\}$, where σ is a variable different from α and β. This entailment does not hold (consider $\rho_1(\alpha) = \top$ and $\rho_1(\beta) = \bot \rightarrow \bot$), but the entailment $\emptyset \models_{\{\alpha,\beta\}} \{\alpha \Rightarrow \sigma\}$ and $\emptyset \models_{\{\alpha,\beta\}} \{\beta \Rightarrow \sigma\}$ both hold.

Terms can be represented as directed trees with nodes labeled with constructors and variables. Term graphs (or term DAGs) are a more compact representation to allow sharing of common sub-terms.

Definition 5.1.6 (Term DAG) In a term DAG, a variable is represented as a node with out-degree 0. A function type is represented as a node $\rightarrow$ with out-degree 2, one for the domain and one for the range. No two different nodes in a term DAG may represent the same term (sharing must be maximal).

We also represent conditional constraints in the term graph. We represent $\alpha \Rightarrow \tau$ as a directed edge from the node representing α to the node representing τ. We call such an edge a *conditional edge*, in contrast to the two outgoing edges from a $\rightarrow$ node, which are called *structural edges*.

5.2 Entailment of equality constraints

In this section we consider the entailment problems for equality constraints. These results are useful in later sections. Algorithms for entailment over equality constraints are known [JM94, ST94, Col82, Col84]. We include them for completeness since these results are used heavily in later sections for studying entailment over conditional equality constraints.

$$\begin{aligned}
\{\tau_1 = \tau_2, \tau_2 = \tau_3\} \subseteq S &\Rightarrow \{\tau_1 = \tau_3\} \subseteq S \\
\{\tau_1 \rightarrow \tau_2 = \tau_1' \rightarrow \tau_2'\} \subseteq S &\Rightarrow \{\tau_1 = \tau_1', \tau_2 = \tau_2'\} \subseteq S \\
\{\bot = \top\} \subseteq S &\Rightarrow \text{not satisfiable} \\
\{\bot = \tau_1 \rightarrow \tau_2\} \subseteq S &\Rightarrow \text{not satisfiable} \\
\{\top = \tau_1 \rightarrow \tau_2\} \subseteq S &\Rightarrow \text{not satisfiable}
\end{aligned}$$

Figure 5.1: Closure rules for equality constraints.

$$\begin{aligned}
\{\tau_1 = \tau_2, \tau_2 = \tau_3\} \subseteq S &\Rightarrow \{\tau_1 = \tau_3\} \subseteq S \\
\{\tau_1 = \tau_1', \tau_2 = \tau_2'\} \subseteq S &\Rightarrow \{\tau_1 \rightarrow \tau_2 = \tau_1' \rightarrow \tau_2'\} \subseteq S
\end{aligned}$$

Figure 5.2: Congruence closure for equality constraints.

5.2.1 Simple entailment

We first consider the simple entailment problem for equality constraints. For $C \models c$ to hold, the constraint c cannot put extra restrictions (beyond those in C) on the variables in C. We use this idea to get an efficient algorithm for deciding whether $C \models c$ for equality constraints.

Recall that the basic algorithm for checking the satisfiability of an equality constraint system is to put the constraint system into some closed form according to the rules in Figure 5.1 (Robinson's algorithm) [Rob71].

An efficient implementation of Robinson's algorithm is based on the union-find data structure. The algorithm operates on a graph representation of the constraints and closes the graphs according to structural decomposition. A union-find data structure maintains equivalence classes, with a designated *representative* for each equivalence class. For any term τ, we denote the equivalence class to which τ belongs by its representative $\textsc{ecr}(\tau)$. If the algorithm succeeds, the resulting DAG forest represents

1. Compute m.g.u. of C. If fail, output YES; else continue.
2. Compute the congruence closure on the m.g.u. of C. If $\alpha = \beta$ is in the closure, output YES; else output NO.

Figure 5.3: Simple entailment $C \vDash \alpha = \beta$ over equality constraints.

the *most general unifier* (m.g.u.) of the original constraint system. The algorithm fails if it discovers a constructor mismatch (the last three rules in Figure 5.1).

The standard unification algorithm is not sufficient for deciding entailment because structural equivalence is not explicit in the resulting DAG representation. That is, constraints of the form $\tau_1 \rightarrow \tau_2 = \tau_1' \rightarrow \tau_2'$ are decomposed into $\tau_1 = \tau_1'$ and $\tau_2 = \tau_2'$, but the equivalence of $\tau_1 \rightarrow \tau_2$ and $\tau_1' \rightarrow \tau_2'$ is not represented explicitly. Moreover, besides constraint decomposition, there are situations in which $\tau_1 = \tau_1'$ and $\tau_2 = \tau_2'$, but the equivalence $\tau_1 \rightarrow \tau_2 = \tau_1' \rightarrow \tau_2'$ is not explicitly represented. For entailment, we would like equivalence classes to mean both that all members of a class X are equal in all solutions, but also that every other equivalence class Y is *different* from X in at least one solution. Thus, equivalence classes should be as large as possible (maximal). This property is guaranteed by congruence closure.

For $C \vDash \alpha = \beta$ to hold, it suffices to check whether $\alpha = \beta$ is implied by C with respect to the congruence closure rules in Figure 5.2, *i.e.*, whether $\alpha = \beta$ is a constraint in the congruence closure of C. Congruence closure can be computed in $\mathcal{O}(n \log n)$ [NO80]. Figure 5.3 gives an algorithm for simple entailment over equality constraints [2].

Lemma 5.2.1 Let C be a constraint system. C and $Cong(C)$, the congruence closure of C, have the same solutions.

Proof. The rules in Figure 5.2 preserve solutions. □

[2] The congruence closure computation is unnecessary. We could simply add the constraint $\alpha = \beta$ to the m.g.u. of C_1 and continue with unification to check if any two distinct equivalence classes are merged. This results in an almost linear time algorithm for a single query. Congruence closure gives a simpler explanation, and also gives an algorithm that answers queries in constant time.

The algorithm clearly runs in $\mathcal{O}(n \log n)$, where $n = |C|$.

Theorem 5.2.2 (Correctness) The algorithm in Figure 5.3 is correct.

Proof. If the algorithm outputs YES, the constraint $\alpha = \beta$ is contained in the congruence closure of the m.g.u. of C. By Lemma 5.2.1, we know that C and the congruence closure of C's m.g.u. have the same solutions. Thus $C \vDash \alpha = \beta$.

If the algorithm outputs NO, then the constraint $\alpha = \beta$ is not in the congruence closure of the m.g.u. of C. One can show with an induction on the structure of the *equivalence class representatives* of α and β, *i.e.*, $\textsc{ecr}(\alpha)$ and $\textsc{ecr}(\beta)$, that any two non-congruent classes admit a valuation $\rho \vDash C$ which maps the two non-congruent classes to different ground types. Thus, $C \nvDash \alpha = \beta$. $\square$

5.2.2 Restricted entailment

In this subsection, we consider restricted entailment for equality constraints. This relation is more involved, since for polymorphic analyses, there are some interface variables we are interested in, and the other internal variables may be eliminated, which can result in a smaller constraint system.

Flanagan and Felleisen [FF97] consider this form of entailment for a class of set constraints and show that the problem to be PSPACE-hard. However the problem has not been considered for other forms of constraints, including equality constraints.

The goal is to decide $C_1 \vDash_E C_2$ for two constraint systems C_1 and C_2 and $E = \mathrm{fv}(C_1) \cup \mathrm{fv}(C_2)$. Recall that $C_1 \vDash_E C_2$ if and only if for every valuation $\rho_1 \vDash C_1$ there exists a valuation $\rho_2 \vDash C_2$ such that $\rho_1(\alpha) = \rho_2(\alpha)$ for all $\alpha \in E$.

We first consider an example to illustrate the issues. Consider the two constraint systems $C_1 = \{\alpha = \sigma \to \sigma\}$ and $C_2 = \{\alpha = \sigma_1 \to \sigma_2\}$ with the interface variables $E = \{\alpha\}$. For any valuation $\rho \vDash C_1$, we know $\rho(\alpha) = \tau \to \tau$ for some τ. Consider the valuation ρ' of C_2 with $\rho'(\alpha) = \rho(\alpha)$ and $\rho'(\sigma_1) = \rho'(\sigma_2) = \rho(\sigma)$. It is easy to see that $\rho' \vDash C_2$, and thus the relation $C_1 \vDash_E C_2$ holds. The algorithm for simple entailment given in Figure 5.3 does not apply since for this example, it would give the answer

1. Compute m.g.u. of C_1. If fail, output YES; else continue.
2. Add each term of C_2 to the m.g.u. of C_1 if the term is not already present.
3. Compute the congruence closure on the term graph obtained in Step 2.
4. Unify the constraints of C_2 in the result obtained in Step 3, and perform the following check. For any two non-congruent classes that are unified, we require at least one of the representatives to be a variable in $\mathrm{fv}(C_2) \setminus E$. If this requirement is not met, output NO; else output YES.

Figure 5.4: Restricted entailment $C_1 \models_E C_2$ over equality constraints.

NO. Note that if $C_1 \models C_2$ (holds iff for all $c \in C_2$ we have $C_1 \models c$) then $C_1 \models_E C_2$ for any E. The converse, however, is not true, as shown by the example.

We modify the simple entailment algorithm over equality constraints to get an algorithm for restricted entailment over equality constraints. The intuition behind the algorithm is that we can relax the requirement of simple entailment to allow internal variables of C_2 to be added to equivalence classes of the term DAG representation of the m.g.u. of C_1, as long as no equivalence classes of C_1 are merged. The algorithm is given in Figure 5.4 [3].

In Figure 5.4, the choice of representatives for equivalence classes is important. We pick representatives in the following order, which guarantees that if the representative is in $\mathrm{fv}(C_2) \setminus E$, then there is no variable in $\mathrm{fv}(C_1)$ or a constructor in the equivalence class:

1. $\bot$, $\top$, and $\rightarrow$ nodes;
2. variables in $\mathrm{fv}(C_1)$;
3. variables in $\mathrm{fv}(C_2) \setminus E$.

[3]The step of congruence closure is again not necessary here.

$$\begin{cases} \rho(\alpha) = \rho_1(\alpha) & \text{if } \alpha \in \mathrm{fv}(C_1) \\ \rho(\alpha) = \begin{cases} \rho_1(\textsc{ecr}(\alpha)) & \text{if } \textsc{ecr}(\alpha) \in \mathrm{fv}(C_1) \\ \bot & \text{if } \textsc{ecr}(\alpha) \in \mathrm{fv}(C_2) \setminus E \\ \bot & \text{if } \textsc{ecr}(\alpha) = \bot \\ \top & \text{if } \textsc{ecr}(\alpha) = \top \\ \rho(\tau_1) \to \rho(\tau_2) & \text{if } \textsc{ecr}(\alpha) = \tau_1 \to \tau_2 \end{cases} & \text{if } \alpha \in \mathrm{fv}(C_2) \setminus E \end{cases}$$

Figure 5.5: Constructed valuation ρ.

Let $n = |C_1|$ and $m = |C_2|$. It is easy to see that the algorithm takes time $\mathcal{O}((m+n)\log(m+n))$.

Theorem 5.2.3 (Correctness) The algorithm in Figure 5.4 is correct.

Proof. Suppose the algorithm outputs YES. If C_1 is not satisfiable then clearly $C_1 \vDash_E C_2$. Let ρ_1 be a satisfying valuation of C_1. Consider the valuation ρ given in Figure 5.5.

The valuation ρ is clearly well-defined. Let ρ_2 denote the valuation obtained by restricting ρ to $\mathrm{fv}(C_2)$, *i.e.*, the variables in C_2. We want to show that $\rho_2 \vDash C_2$. Since the algorithm outputs YES, when adding the constraints in C_2, the only change to the graph is adding variables in $\mathrm{fv}(C_2) \setminus E$ to some existing equivalence classes. By the construction of ρ, one can see that ρ satisfies all the induced constraints in the term graph at step 4 of the algorithm. Thus, we have $\rho \vDash C_1 \cup C_2$, and therefore $\rho_2 \vDash C_2$. Hence, we have $C_1 \vDash_E C_2$.

Conversely, suppose the algorithm outputs NO. Then there exist two equivalence classes to be unified neither of whose ECR is a variable in $\mathrm{fv}(C_2) \setminus E$. There are two cases.

- In the first case, one ECR is a variable in $\mathrm{fv}(C_1)$, say α. If the other representative is $\bot$, then any valuation $\rho_1 \vDash C_1$ with $\rho_1(\alpha) = \top$ gives a witness for $C_1 \nvDash_E C_2$. The case where the other representative is $\top$ or a $\to$ node is similar. If the other

Let C be a system of constraints. The following algorithm outputs a term graph representing the solutions of C.

1. Let G be the term graph CONDRESOLVE(C).
2. For each variable α in $\mathrm{fv}(C)$, check whether it must be $\perp$: If neither $G \cup \{\alpha = \top\}$ nor $G \cup \{\alpha = \sigma_1 \to \sigma_2\}$ is satisfiable, add $\alpha = \perp$ to G.

Figure 5.6: Modified conditional unification algorithm.

representative is a variable $\beta \in \mathrm{fv}(C_1)$, any valuation $\rho_1 \models C_1$ with $\rho_1(\alpha) = \top$ and $\rho_1(\beta) = \perp$ is a witness for $C_1 \not\models_E C_2$.

- In the second case, both ECRs are constructors. If there is a constructor mismatch, then $C_1 \cup C_2$ is not satisfiable. Since C_1 is satisfiable, then $C_1 \not\models_E C_2$ (Since if $C_1 \models_E C_2$, any satisfying valuation for C_1 can be extended to a satisfying valuation for C_2.)

  Note that if there is no constructor mismatch (where both representatives are $\to$ nodes), the error is detected when trying to unify the terms represented by these two nodes. Thus it falls into one of the above cases.

Thus it follows that $C_1 \not\models_E C_2$.

□

5.3 Conditional equality constraints

In this section, we consider the two entailment problems for constraint systems with conditional equality constraints. Recall for $\alpha \Rightarrow \tau$ to be satisfied by a valuation ρ, either $\rho(\alpha) = \perp$ or $\rho(\alpha) = \rho(\tau)$.

Lemma 5.3.1 (Transitivity of $\Rightarrow$) Any valuation ρ satisfying $\alpha \Rightarrow \beta$ and $\beta \Rightarrow \gamma$, also satisfies $\alpha \Rightarrow \gamma$.

1. (a) Run the conditional unification algorithm in Figure 5.6 on $C \cup \{\alpha = \top\}$. If not satisfiable, then SUCCESS; else continue.

   (b) Compute strongly connected components (SCC) on the conditional edges and merge the nodes in every SCC. This step yields a modified term graph.

   (c) Compute congruence closure on the term graph obtained in Step 1b. We do not consider the conditional edges for computing congruence closure.

   (d) If $\beta = \top$ is in the closure, SUCCESS; else FAIL.

2. (a) Run the conditional unification algorithm in Figure 5.6 on $C \cup \{\alpha = \sigma_1 \to \sigma_2\}$, where σ_1 and σ_2 are two fresh variables not in $\mathrm{fv}(C) \cup \{\alpha, \beta\}$. If not satisfiable, then SUCCESS; else continue.

   (b) Compute strongly connected components (SCC) on the conditional edges and merge the nodes in every SCC. This step yields a modified term graph.

   (c) Compute congruence closure on the term graph obtained in Step 2b. Again, we do not consider the conditional edges for computing congruence closure.

   (d) If $\beta = \sigma_1 \to \sigma_2$ is in the closure, SUCCESS; else FAIL.

3. If both cases return SUCCESS, output YES; else output NO.

Figure 5.7: Simple entailment $C \models \alpha \Rightarrow \beta$ over conditional equality constraints.

Consider the constraints $\{\alpha \Rightarrow \top, \alpha \Rightarrow \bot \to \bot\}$. The only solution is $\alpha = \bot$. The fact that α must be $\bot$ is not explicit. For entailment, we want to make the fact that α must be $\bot$ explicit.

Assume that we have run CondResolve on the constraints to get a term graph G. For each variable α, we check whether it must be $\bot$. If both adding $\alpha = \top$ to

G and $\alpha = \sigma_1 \to \sigma_2$ to G (for fresh variables σ_1 and σ_2) fail, α must be $\bot$, in which case, we add $\alpha = \bot$ to G. We repeat this process for each variable. Notice that this step can be done in polynomial time. We present this modification to the conditional unification algorithm in Figure 5.6.

5.3.1 Simple entailment

In this subsection, we present an algorithm for deciding $C \vDash \alpha = \beta$ and $C \vDash \alpha \Rightarrow \beta$ where C_1 and C_2 are constraint systems with conditional equality constraints. Note $C \vDash \alpha = \beta$ holds if and only if $C \vDash \alpha \Rightarrow \beta$ and $C \vDash \beta \Rightarrow \alpha$ both hold. We give the algorithm in Figure 5.7. The basic idea is that to check $C \vDash \alpha \Rightarrow \beta$ holds we have two cases: when α is $\top$ and when α is a function type. In both cases, we require $\beta = \alpha$. The problem then basically reduces to simple entailment over equality constraints. As for entailment for equality constraints, congruence closure is required to make explicit the implied equalities between terms involving $\to$. Computing strong components is used to make explicit, for example, $\alpha = \beta$ if both $\alpha \Rightarrow \beta$ and $\beta \Rightarrow \alpha$. It is easy to see that the algorithm runs in worst case polynomial time in the size of C.

Theorem 5.3.2 The simple entailment algorithm in Figure 5.7 is correct.

Proof. Suppose that the algorithm outputs YES. Let ρ be a satisfying valuation for C. We have three cases.

- If $\rho(\alpha) = \bot$, then $\rho \vDash \alpha \Rightarrow \beta$.
- If $\rho(\alpha) = \top$, then $\rho \vDash C \cup \{\alpha = \top\}$. Thus $C \cup \{\alpha = \top\}$ is satisfiable. We then have $\beta = \top$ is in the closure of $C \cup \{\alpha = \top\}$. Hence, $\rho \vDash \beta = \top$, which implies that $\rho \vDash \alpha \Rightarrow \beta$.
- If $\rho(\alpha) = \tau_1 \to \tau_2$ where τ_1 and τ_2 are ground types, we have $C \cup \{\alpha = \sigma_1 \to \sigma_2\}$ is satisfiable. We thus have $\beta = \sigma_1 \to \sigma_2$ is in the closure of $C \cup \{\alpha = \sigma_1 \to \sigma_2\}$. Hence $\rho \vDash \beta = \sigma_1 \to \sigma_2$, and which implies that $\rho \vDash \alpha \Rightarrow \beta$.

Combining the three cases, we conclude that $C \vDash \alpha \Rightarrow \beta$.

Suppose the algorithm outputs NO. Then at least one of the two cases returns FAIL. Assume that the case with $\alpha = \top$ returns FAIL. Then $\beta = \top$ is not in the congruence closure of $C \cup \{\alpha = \top\}$. By assigning all the remaining *conditional variables* to $\bot$ (variables appearing as the antecedent of the conditional constraints) in the graph, we can exhibit a witness valuation ρ that satisfies C but does not satisfy $\alpha \Rightarrow \beta$ (same as the case for simple entailment over equality constraints). The other case where $\alpha = \sigma_1 \to \sigma_2$ is similar. Hence $C \nvDash \alpha \Rightarrow \beta$.

$\square$

5.3.2 Restricted entailment over atomic constraints

Before considering the problem $C_1 \vDash_E C_2$ over conditional equality constraints, we look at a simpler case, in which all the constraints are between variables. With minor modifications, the presented algorithm can handle constraints over atoms (variables, $\bot$ and $\top$).

We first characterize the solutions of a constraint system with respect to a set of variables. Let C be a constraint system and $E \subseteq \mathrm{fv}(C)$. Recall that a conditional constraint $\alpha \Rightarrow \beta$ is represented as a directed edge from the node representing α to the node representing β. We compute the strongly connected components (SCC) of the term graph representation of the conditional constraints. We now perform the following transformations:

- If a variable $\alpha \notin E$ appears in a strong component with variables in E, then remove α and its incident edges from the component.
- Remove any conditional edge if the antecedent component consists only of variables in $\mathrm{fv}(C) \setminus E$.
- Remove any isolated component consisting only of variables in $\mathrm{fv}(C) \setminus E$.

Compute the transitive closure relation on the resulting dag. The resulting graph is the *normal form* $\mathrm{NF}(C^E)$ of C with respect to E.

Example 11 Consider the constraints

$$\{\tau_0 \Rightarrow \alpha, \alpha \Rightarrow \tau_1, \beta \Rightarrow \tau_2, \tau_2 \Rightarrow \tau_3, \tau_3 \Rightarrow \beta, \tau_3 \Rightarrow \tau_1\}$$

The graph representation of the constraints is:

$$\begin{tikzcd}
\tau_0 \arrow[dr] & & \beta \arrow[rr] & & \tau_2 \arrow[dl] \\
& \alpha \arrow[dr] & & \tau_3 \arrow[ul] \arrow[dl] & \\
& & \tau_1 & &
\end{tikzcd}$$

The graph $\text{NF}(C^{\{\alpha,\beta\}})$ is

$$\begin{tikzcd}
\alpha \arrow[dr] & & \beta \arrow[dl] \\
& \tau_1 &
\end{tikzcd}$$

The solutions w.r.t. $\{\alpha, \beta\}$ are the same, either α is $\perp$, or β is $\perp$, or $\alpha = \beta$.

For a constraint system C, we denote by $S(C) \mid_E$ the set of satisfying valuations restricted to E, i.e.

$$S(C) \mid_E = \{\rho' \mid \rho \vDash C \text{ and } \rho' = \rho \mid_E\},$$

where $\rho \mid_E$ denotes the valuation of ρ restricted to the variables E.

Lemma 5.3.3 $S(C) \mid_E = S(\text{NF}(C^E)) \mid_E$.

Proof. By transitivity of $\Rightarrow$ and the fact that we only add transitive constraints and remove other constraints from C to get $\text{NF}(C^E)$, it is clear that $S(C) \mid_E \subseteq S(\text{NF}(C^E)) \mid_E$.

For the other direction, we need to show that each valuation in $S(\text{NF}(C^E)) \mid_E$ is also in $S(C) \mid_E$. Let ρ be a valuation in $S(\text{NF}(C^E)) \mid_E$. It is extensible to a valuation ρ' of $\text{NF}(C^E)$. We want to show that ρ' can be extended to a valuation ρ'' that satisfies C. We define ρ'' as follows

$$\begin{cases} \rho''(\alpha) = \rho'(\alpha) & \text{if } \alpha \in \text{fv}(\text{NF}(C^E)) \\ \rho''(\alpha) = \rho'(\beta) & \text{if } \text{ECR}(\alpha) = \text{ECR}(\beta) \text{ and } \beta \in E \\ \rho''(\alpha) = \perp & \text{otherwise} \end{cases}$$

$$\begin{tikzcd}
\alpha \arrow[d] \arrow[dr] & \beta \arrow[dl] \arrow[dr] & \gamma \arrow[d] \arrow[dl] \\
\tau_1 & \tau_2 & \tau_3
\end{tikzcd}$$

(a) Term graph for C_1.

$$\begin{tikzcd}
\alpha \arrow[dr] & \beta \arrow[d] & \gamma \arrow[dl] \\
 & \tau &
\end{tikzcd}$$

(b) Term graph for C_2.

Figure 5.8: Example.

where $\text{ECR}(\alpha) = \text{ECR}(\beta)$ means that α and β are in the same strong component.

One can verify that $\rho'' \vDash C$ by observing that the only constraints that are removed from C are of the forms $\tau \Rightarrow \alpha$, $\tau_1 = \tau_2$, or $\tau = \beta$ where τ, τ_1, and τ_2 are variables not in E and α and β are in E. These constraints are satisfied by our construction of ρ''.

$\square$

After this transformation only interface variables in E can appear as the left-side of conditional constraints.

We now consider the decision problem $C_1 \vDash_E C_2$. Before giving the algorithm, let us look at another example. Consider the constraints

$$C_1 = \{\alpha \Rightarrow \tau_1, \beta \Rightarrow \tau_1, \alpha \Rightarrow \tau_2, \gamma \Rightarrow \tau_2, \beta \Rightarrow \tau_3, \gamma \Rightarrow \tau_3\}$$

and

$$C_2 = \{\alpha \Rightarrow \tau, \beta \Rightarrow \tau, \gamma \Rightarrow \tau\}$$

We want to determine that $C_1 \equiv_E C_2$ where $E = \{\alpha, \beta, \gamma\}$. The term graph representations for C_1 and C_2 are given in Figure 5.8. These are also their normal forms. Notice that the constraints $\alpha \Rightarrow \tau_1$ and $\beta \Rightarrow \tau_1$ force $\alpha = \beta$ when $\alpha \neq \bot$ and $\beta \neq \bot$. Thus we can easily characterize C_1's solutions restricted to E as

- All of α, β, and γ are $\bot$;
- One of them is not $\bot$ and can be any (non-$\bot$) value;
- Two of them are not $\bot$ and have the same (non-$\bot$) value ;
- All three have the same (non-$\bot$) value.

1. Compute $\text{NF}(C_1^E)$ and $\text{NF}(C_2^E)$;
2. Perform the following checks (where $\alpha \in E$ and $\beta \in E$, and $\tau \notin E$ and $\tau' \notin E$)
   (a) if $(\alpha = \beta) \in \text{NF}(C_2^E)$, check $(\alpha = \beta) \in \text{NF}(C_1^E)$;
   (b) if $(\alpha \Rightarrow \beta) \in \text{NF}(C_2^E)$, check $(\alpha = \beta) \in \text{NF}(C_1^E)$ or $(\alpha \Rightarrow \beta) \in \text{NF}(C_1^E)$;
   (c) if $\{\alpha \Rightarrow \tau, \beta \Rightarrow \tau\} \subseteq \text{NF}(C_2^E)$, check $(\alpha = \beta) \in \text{NF}(C_1^E)$ or $\{\alpha \Rightarrow \tau', \beta \Rightarrow \tau'\} \subseteq \text{NF}(C_1^E)$;
3. if any of these tests fail, output NO; else output YES.

Figure 5.9: Algorithm for restricted entailment over atomic constraints.

Similarly, we get the same characterization for C_2, thus $C_1 \equiv_E C_2$.

We now give our algorithm for deciding $C_1 \vDash_E C_2$. The algorithm is in Figure 5.9. Case 2c handles the tricky case illustrated by Figure 5.8.

We first analyze the running time of the algorithm. The time to compute $\text{NF}(C_1^E)$ and $\text{NF}(C_2^E)$ is $\mathcal{O}(m^2 + n^2)$ where $m = |C_1|$ and $n = |C_2|$. The time to perform the checks can be done in time $\mathcal{O}(mn)$. Thus the running time of the algorithm is $\mathcal{O}(m^2 + n^2 + mn)$.

Theorem 5.3.4 The algorithm in Figure 5.9 is correct.

Proof. Let $\mathit{pre}^C(\alpha)$ be $\{\beta \mid \beta \Rightarrow \alpha\}$ in constraints C. Define $T_\rho^C(\alpha) = \bot$ if the valuation ρ maps all variables in $\mathit{pre}^C(\alpha)$ to $\bot$, otherwise, $T_\rho^C(\alpha) = \rho(\beta)$ for any $\beta \in \mathit{pre}^C(\alpha)$ with $\rho(\beta) \neq \bot$.

Assume the algorithm outputs YES. We show that for each valuation $\rho \vDash \text{NF}(C_1^E)$, the valuation $\rho|_E$ can be extended to a valuation ρ' that satisfies $\text{NF}(C_2^E)$. We define ρ' as follows

$$\begin{cases} \rho'(\alpha) = \rho(\alpha) & \text{if } \alpha \in E \\ \rho'(\alpha) = T_\rho^{\text{NF}(C_2^E)}(\alpha) & \text{if } \alpha \in \text{fv}(\text{NF}(C_2^E)) \setminus E \end{cases}$$

We first show that ρ' is well-defined. We argue that all variables in $pre^{\mathrm{NF}(C_2^E)}(\alpha)$ must be mapped to the same value. Let α_1 and α_2 be in $pre^{\mathrm{NF}(C_2^E)}(\alpha)$ and $\rho(\alpha_1) \neq \bot$ and $\rho(\alpha_2) \neq \bot$. Since $\alpha_1 \Rightarrow \alpha$ and $\alpha_2 \Rightarrow \alpha$ are constraints in $\textsc{nf}(C_2^E)$, we have either $(\alpha_1 = \alpha_2) \in \textsc{nf}(C_1^E)$ or $\{\alpha_1 \Rightarrow \alpha', \alpha_2 \Rightarrow \alpha'\} \subseteq \textsc{nf}(C_1^E)$. Then we have $\rho(\alpha_1) = \rho(\alpha_2)$. Thus ρ' is well-defined. To see that $\rho' \vDash \textsc{nf}(C_2^E)$, notice that ρ' satisfies each constraint in $\textsc{nf}(C_2^E)$, *i.e.*, $\rho' \vDash \alpha = \beta$, $\rho' \vDash \alpha \Rightarrow \beta$, and $\rho' \vDash \alpha \Rightarrow \tau$. Thus $S(\textsc{nf}(C_1^E)) \mid_E \subseteq S(\textsc{nf}(C_2^E)) \mid_E$. By Lemma 5.3.3, we have $C_1 \vDash_E C_2$.

If the algorithm outputs NO, at least one of the checks fails in step 2 of the algorithm. We consider the three cases separately.

Case 1. Suppose $\alpha = \beta \in \textsc{nf}(C_2^E)$ but $\alpha = \beta \notin \textsc{nf}(C_1^E)$. Either $\{\alpha \mapsto \top, \beta \mapsto \bot\}$ or $\{\alpha \mapsto \bot, \beta \mapsto \top\}$ can be extended to a valuation $\rho \vDash \textsc{nf}(C_1^E)$. However, clearly $\rho \mid_E$ cannot be extended to a $\rho' \vDash \textsc{nf}(C_2^E)$. Thus $C_1 \vDash_E C_2$ does not hold.

Case 2. Suppose $\alpha \Rightarrow \beta \in \textsc{nf}(C_2^E)$, and $\alpha = \beta \notin \textsc{nf}(C_1^E)$ and $\alpha \Rightarrow \beta \notin \textsc{nf}(C_1^E)$. One can show that $\{\alpha \mapsto \top, \beta \mapsto \bot\}$ can be extended to a valuation $\rho \vDash \textsc{nf}(C_1^E)$. Thus, $C_1 \vDash_E C_2$ does not hold.

Case 3. Suppose $\{\alpha \Rightarrow \tau, \beta \Rightarrow \tau\} \subseteq \textsc{nf}(C_2^E)$, and $(\alpha = \beta) \notin \textsc{nf}(C_1^E)$ and $\{\alpha \Rightarrow \tau', \beta \Rightarrow \tau'\} \nsubseteq \textsc{nf}(C_1^E)$ for any $\tau' \notin E$. The partial valuation $\{\alpha \mapsto \top, \beta \mapsto \bot \to \bot\}$ can be extended to a valuation ρ that satisfies C_1. In particular, we construct ρ as follows

$$\begin{cases} \rho(\alpha) = \top & \\ \rho(\beta) = \bot \to \bot & \\ \rho(\gamma) = \rho(\alpha) & \text{if } \textsc{ecr}(\gamma) = \textsc{ecr}(\alpha) \\ \rho(\gamma) = \rho(\alpha) & \text{if } \textsc{ecr}(\alpha) \Rightarrow \textsc{ecr}(\gamma) \\ \rho(\gamma) = \rho(\beta) & \text{if } \textsc{ecr}(\beta) = \textsc{ecr}(\gamma) \\ \rho(\gamma) = \rho(\beta) & \text{if } \textsc{ecr}(\beta) \Rightarrow \textsc{ecr}(\gamma) \\ \rho(\gamma) = \bot & \text{otherwise} \end{cases}$$

where ECR is defined as the representative of a SCC. In choosing a ECR for a SCC, the variables in E have precedence over the internal variables.

One can show that the constructed valuation ρ satisfies C_1. However $\{\alpha \mapsto \top, \beta \mapsto \bot \to \bot\}$ cannot be extended to a valuation that satisfies C_2 since this would require

$$
\begin{aligned}
S(\{\alpha_1 \Rightarrow \bot, \alpha_2 \Rightarrow \sigma_1\}) &= \{\alpha_1 \mapsto \bot\} && (5.1)\\
S(\{\alpha_2 \Rightarrow \sigma_1, \alpha_2 \Rightarrow \sigma_2\}) &= S^* && (5.2)\\
S(\{\alpha_3 \Rightarrow \sigma_2, \alpha_3 \Rightarrow \top\}) &= \{\alpha_3 \mapsto \bot\} \cup \{\alpha_3 \mapsto \top\} && (5.3)\\
S(\{\alpha_1 \Rightarrow \bot, \alpha_2 \Rightarrow \sigma_1\}) &= \{\alpha_1 \mapsto \bot\} && (5.4)\\
S(\{\alpha_1 \Rightarrow \bot, \alpha_2 \Rightarrow \sigma_2\}) &= \{\alpha_1 \mapsto \bot\} && (5.5)\\
S(\{\alpha_1 \Rightarrow \bot, \alpha_3 \Rightarrow \sigma_2\}) &= \{\alpha_1 \mapsto \bot\} && (5.6)\\
S(\{\alpha_1 \Rightarrow \bot, \alpha_3 \Rightarrow \top\}) &= \{\alpha_1 \mapsto \bot, \alpha_3 \mapsto \bot\} \cup \{\alpha_1 \mapsto \bot, \alpha_3 \mapsto \top\} && (5.7)\\
S(\{\alpha_1 \Rightarrow \sigma_1, \alpha_2 \Rightarrow \sigma_1\}) &= \{\alpha_1 \mapsto \bot\} \cup \{\alpha_2 \mapsto \bot\} \cup S(\{\alpha_1 = \alpha_2\}) && (5.8)\\
S(\{\alpha_1 \Rightarrow \sigma_1, \alpha_2 \Rightarrow \sigma_2\}) &= S^* && (5.9)\\
S(\{\alpha_1 \Rightarrow \sigma_1, \alpha_3 \Rightarrow \sigma_2\}) &= S^* && (5.10)\\
S(\{\alpha_1 \Rightarrow \sigma_1, \alpha_3 \Rightarrow \top\}) &= \{\alpha_3 \mapsto \bot\} \cup \{\alpha_3 \mapsto \top\} && (5.11)\\
S(\{\alpha_2 \Rightarrow \sigma_1, \alpha_3 \Rightarrow \sigma_2\}) &= S^* && (5.12)\\
S(\{\alpha_2 \Rightarrow \sigma_1, \alpha_3 \Rightarrow \top\}) &= \{\alpha_3 \mapsto \bot\} \cup \{\alpha_3 \mapsto \top\} && (5.13)\\
S(\{\alpha_2 \Rightarrow \sigma_2, \alpha_3 \Rightarrow \sigma_2\}) &= \{\alpha_2 \mapsto \bot\} \cup \{\alpha_3 \mapsto \bot\} \cup S(\{\alpha_2 = \alpha_3\}) && (5.14)\\
S(\{\alpha_2 \Rightarrow \sigma_2, \alpha_3 \Rightarrow \top\}) &= \{\alpha_3 \mapsto \bot\} \cup \{\alpha_3 \mapsto \top\} && (5.15)
\end{aligned}
$$

Figure 5.10: Solutions for all subsets of two constraints.

unifying $\top$ with $\bot \to \bot$.

$\square$

5.4 Restricted entailment over conditional equality constraints

In this section, we give a polynomial time algorithm for restricted entailment over conditional constraints. For simplicity, we consider the language without $\top$. The modified language is given by the following grammar

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

With some extra checks, the presented algorithm can be adapted to include $\top$ in the language.

Consider the following example.

Example 12

$$\begin{tikzcd}[column sep=small] & \alpha_1 \arrow[dl] \arrow[dr] & & \alpha_2 \arrow[dl] \arrow[dr] & & \alpha_3 \arrow[dl] \arrow[dr] & \\ \bot & & \sigma_1 & & \sigma_2 & & \top \end{tikzcd}$$

Notice that the solutions of the constraints in Example 12 with respect to $\{\alpha_1, \alpha_2, \alpha_3\}$ are

$$\begin{aligned} \{\alpha_1 \mapsto \bot, \alpha_3 \mapsto \bot\} & \quad \cup \\ \{\alpha_1 \mapsto \bot, \alpha_2 \mapsto \bot, \alpha_3 \mapsto \top\} & \quad \cup \\ \{\alpha_1 \mapsto \bot, \alpha_2 \mapsto \top, \alpha_3 \mapsto \top\} & \end{aligned}$$

Now suppose we do the following: we take pairs of constraints, find their solutions with respect to $\{\alpha_1, \alpha_2, \alpha_3\}$, and take the intersection of the solutions. Let S^* denote the set of all valuations. Figure 5.10 shows the solutions for all the subsets of two constraints with respect to $\{\alpha_1, \alpha_2, \alpha_3\}$. One can show that the intersection of these solutions is the same as the solution for all the constraints. Intuitively, the solutions of a system of conditional constraints can be characterized by considering all pairs of constraints independently. We can make this intuition formal by putting some additional requirements on the constraints.

Here is the route we take to develop a polynomial time algorithm for restricted entailment over conditional constraints.

Section 5.4.1

We introduce a notion of a *closed system* and show that closed systems have the property that it is sufficient to consider pairs of conditional constraints in determining the solutions of the complete system with respect to the interface variables.

Section 5.4.2

We show that restricted entailment with a pair of conditional constraints can be decided in polynomial time, *i.e.*, $C \models_E C_= \cup \{c_1, c_2\}$ can be decided in polynomial time, where $C_=$ consists of equality constraints, and c_1 and c_2 are conditional constraints.

Section 5.4.3

We show how to reduce restricted entailment to restricted entailment in terms of closed systems. In particular, we show how to reduce $C_1 \models_E C_2$ to $C_1' \models_{E'} C_2'$ where C_2' is closed.

Combining the results, we arrive at a polynomial time algorithm for restricted entailment over conditional constraints.

5.4.1 Closed systems

We define the notion of a closed system and show the essential properties of closed systems for entailment. Before presenting the definitions, we first demonstrate the idea with the example in Figure 5.11. Let C denote the constraints in this example, with α and β the interface variables, and σ, σ_1, and σ_2 the internal variables. The intersection of the solutions of all the pairs of constraints is: α is either $\bot$ or $\tau \to \bot$ and β is either $\bot$ or $\tau' \to \bot$ for some τ and τ'. However, the solutions of C require that if $\alpha = \tau \to \bot$ and $\beta = \tau' \to \bot$, and both τ and τ' are non-$\bot$, then $\tau = \tau'$, *i.e.*, $\alpha = \beta$. Thus the intersection of solutions of pairs of constraints contains more valuations than the solution set of the entire system. The reason is that when we consider the set $\{\sigma_1 \Rightarrow \sigma, \sigma_2 \Rightarrow \sigma\}$, the solutions w.r.t. $\{\alpha, \beta\}$ are all valuations. We lose the information that α and β need to be the same in their domain.

We would like to consider σ_1 and σ_2 as interface variables if $\sigma_1 \neq \bot \neq \sigma_2$. We introduce some constraints and new interface variables into the system to *close* it. The modified constraint system is shown in Figure 5.11b. To make explicit the relationship between α and β, two variables α_1 and β_1 (interface variables corresponding to σ_1 and σ_2, respectively) are created with the constraints $\alpha_1 \Rightarrow \sigma$ and $\beta_1 \Rightarrow \sigma$. With this

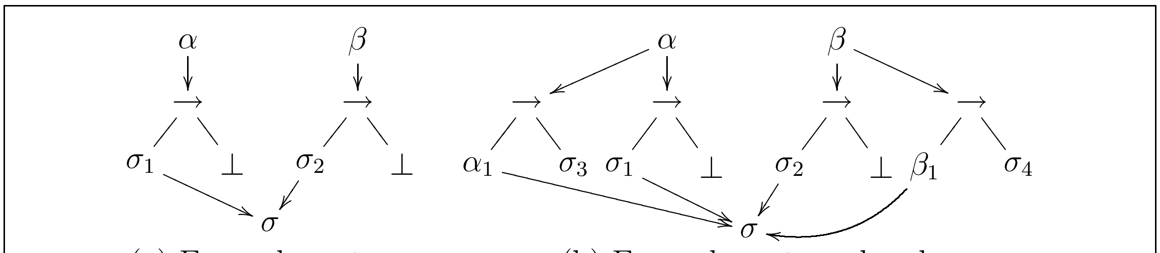


Figure 5.11: An example constraint system and its closed system.

modification, the intersection of solutions of pairs of constraints w.r.t. $\{\alpha, \beta, \alpha_1, \beta_1\}$ is the same as the solution of the modified system. Restricting this intersection w.r.t. $\{\alpha, \beta\}$ we get the solution of the original constraint system. We next show how to systematically close a constraint system.

Definition 5.4.1 (TR) Consider a constraint $\alpha \Rightarrow \tau$ with the variable σ a *proper* subexpression of τ. We define a transformation TR on $\alpha \Rightarrow \tau$ over the structure of τ

- $\textsc{tr}(\sigma, \alpha \Rightarrow \sigma \rightarrow \tau') = \{\alpha \Rightarrow \alpha_1 \rightarrow \sigma_1\}$;

- $\textsc{tr}(\sigma, \alpha \Rightarrow \tau' \rightarrow \sigma) = \{\alpha \Rightarrow \sigma_2 \rightarrow \alpha_2\}$;

- $\textsc{tr}(\sigma, \alpha \Rightarrow \tau_1 \rightarrow \tau_2) =$
$$\begin{cases} \{\alpha \Rightarrow \alpha_1 \rightarrow \sigma_1\} \cup \textsc{tr}(\sigma, \alpha_1 \Rightarrow \tau_1) & \text{if } \sigma \in \text{fv}(\tau_1) \\ \{\alpha \Rightarrow \sigma_2 \rightarrow \alpha_2\} \cup \textsc{tr}(\sigma, \alpha_2 \Rightarrow \tau_2) & \text{otherwise} \end{cases}$$
Note if σ appears in both τ_1 and τ_2, TR is applied only to the occurrence of σ in τ_1.

- $\textsc{tr}(\sigma, \alpha = \tau) = \textsc{tr}(\sigma, \alpha \Rightarrow \tau)$.

The newly created α_i's are called *auxiliary variables*. The variables α_i in the first two cases are called the *matching variable for σ*. The variable α is called the *root* of α_i, and is denoted by $\textsc{root}(\alpha_i)$.

For each auxiliary variable α_i, we denote by $C_{\text{TR}}(\alpha_i)$ the TR constraints accumulated till α_i is created.

Putting this definition to use on the constraint system in Figure 5.11a, $\text{TR}(\sigma_1, \alpha \Rightarrow \sigma_1)$ yields the constraint $\alpha \Rightarrow \alpha_1 \to \sigma_3$ (shown in Figure 5.11b).

To understand the definition of $C_{\text{TR}}(\alpha_i)$, consider $\text{TR}(\sigma, \alpha \Rightarrow ((\sigma \to \bot) \to \bot)) = \{\alpha \Rightarrow \alpha_1 \to \sigma_1, \alpha_1 \Rightarrow \alpha_2 \to \sigma_2\}$, where α_1 and α_2 are the auxiliary variables. We have

$$C_{\text{TR}}(\alpha_1) = \{\alpha \Rightarrow \alpha_1 \to \sigma_1\}$$

and

$$C_{\text{TR}}(\alpha_2) = \{\alpha \Rightarrow \alpha_1 \to \sigma_1, \alpha_1 \Rightarrow \alpha_2 \to \sigma_2\}.$$

Definition 5.4.2 (Closed systems) A system of conditional constraints C is *closed* w.r.t. a set of variables E in C after the following steps:

1. Let $C' = \textsc{CondResolve}(C)$.
2. Set W to E.
3. For each variable $\alpha \in W$, if $\alpha \Rightarrow \tau$ is in C', where $\sigma \in \text{fv}(\tau)$, and $\sigma \Rightarrow \tau' \in C'$, add $\text{TR}(\sigma, \alpha \Rightarrow \tau)$ to C'. Let α' be the matching variable for σ and add $\alpha' \Rightarrow \tau'$ to C'.
4. Set W to the set of auxiliary variables created in Step 3 and repeat Step 3 until W is empty.

Step 3 of this definition warrants explanation. In the example $\text{TR}(\sigma_1, \alpha \Rightarrow \sigma_1)$ we add the constraint $\alpha \Rightarrow \alpha_1 \to \sigma_3$ with α_1 as the matching variable for σ_1. We want to ensure that α_1 and σ_1 are actually the same, so we add the constraint $\alpha_1 \Rightarrow \sigma$. This process must be repeated to expose all such internal variables (such as σ_1 and σ_2).

Next we give the definition of a *forced variable*. Given a valuation ρ for the interface variables, if an internal variable σ is determined already by ρ, then σ is *forced by* ρ. For example, in Figure 5.11, if α is non-$\bot$, then the value of σ_1 is forced by α.

Definition 5.4.3 (Forced variables) We say that an internal variable σ is *forced* by a valuation ρ if any one of the following holds (A is the set of auxiliary variables)

- $\textsc{ecr}(\sigma) = \perp$;
- $\textsc{ecr}(\sigma) = \alpha$, where $\alpha \in E \cup A$;
- $\textsc{ecr}(\sigma) = \tau_1 \to \tau_2$;
- $\rho(\alpha) \neq \perp$ and $\alpha \Rightarrow \tau$ is a constraint where $\sigma \in \mathrm{fv}(\tau)$ and $\alpha \in E \cup A$;
- σ' is forced by ρ to a non-$\perp$ value and $\sigma' \Rightarrow \tau$ is a constraint where $\sigma \in \mathrm{fv}(\tau)$.

Theorem 5.4.4 Let C be a closed system of constraints w.r.t. a set of interface variables E, and let A be the set of auxiliary variables of C. Let $C_=$ and $C_\Rightarrow$ be the systems of equality constraints and conditional constraints respectively. Then

$$S(C) \mid_{E \cup A} = \bigcap_{c_i, c_j \in C_\Rightarrow} S(C_= \cup \{c_i, c_j\}) \mid_{E \cup A} .$$

In other words, it suffices to consider pairs of conditional constraints in determining the solutions of a closed constraint system.

Proof. Since C contains all the constraints in $C_= \cup \{c_i, c_j\}$ for all i and j, thus it follows that

$$S(C) \mid_{E \cup A} \subseteq \bigcap_{c_i, c_j \in C_\Rightarrow} S(C_= \cup \{c_i, c_j\}) \mid_{E \cup A} .$$

It remains to show

$$S(C) \mid_{E \cup A} \supseteq \bigcap_{c_i, c_j \in C_\Rightarrow} S(C_= \cup \{c_i, c_j\}) \mid_{E \cup A} .$$

Let ρ be a valuation in $\bigcap_{c_i, c_j \in C_\Rightarrow} S(C_= \cup \{c_i, c_j\}) \mid_{E \cup A}$. It suffices to show that ρ can be extended to a satisfying valuation ρ' for C. To show this, it suffices to find an extension ρ' of ρ for C such that $\rho' \vDash C_= \cup \{c_i, c_j\}$ for all i and j.

Consider the valuation ρ' obtained from ρ by mapping all the internal variables not forced by ρ (in C) to $\perp$. The valuation ρ' can be uniquely extended to satisfy C if for any c_i and c_j, c_i' and c_j', if σ is forced by ρ in both $C_= \cup \{c_i, c_j\}$ and $C_= \cup \{c_i', c_j'\}$, then it is forced to the same value in both systems. The value that σ is forced to by ρ is denoted by $\rho^!(\sigma)$.

We prove by cases (cf. Definition 5.4.3) that if σ is forced by ρ, it is forced to the same value in pairs of constraints. Let $C_{i,j}$ denote $C_= \cup \{c_i, c_j\}$ and $C_{i',j'}$ denote $C_= \cup \{c'_i, c'_j\}$.

- If $\text{ECR}(\sigma) = \bot$, then σ is forced to the same value, i.e., $\bot$, because $\sigma = \bot \in C_=$.
- If $\text{ECR}(\sigma) = \alpha$, with $\alpha \in E \cup A$, then σ is forced to $\rho(\alpha)$ in both systems, because $\sigma = \alpha \in C_=$.
- If $\text{ECR}(\sigma) = \tau_1 \rightarrow \tau_2$, one can show that ρ forces σ to the same value with an induction over the structure of $\text{ECR}(\sigma)$ (with the two cases above as base cases).
- Assume σ is forced in $C_{i,j}$ because $\alpha \Rightarrow \tau_1 \in C_{i,j}$ with $\rho(\alpha) \neq \bot$ and forced in $C_{i',j'}$ because $\beta \Rightarrow \tau_2 \in C_{i',j'}$ with $\rho(\beta) \neq \bot$. For each extension ρ_1 of ρ with $\rho_1 \models C_{i,j}$, and for each extension ρ_2 of ρ with $\rho_2 \models C_{i',j'}$, we have

  $$\begin{array}{rclcl} \rho(\alpha) & = & \rho_1(\alpha) & = & \rho_1(\tau_1) \\ \rho(\beta) & = & \rho_2(\beta) & = & \rho_2(\tau_2) \end{array}$$

  Consider the constraint system $C_= \cup \{\alpha \Rightarrow \tau_1, \beta \Rightarrow \tau_2\}$. The valuation ρ can be extended to ρ_3 with $\rho_3 \models C_= \cup \{\alpha \Rightarrow \tau_1, \beta \Rightarrow \tau_2\}$. Thus we have

  $$\begin{array}{rclcl} \rho(\alpha) & = & \rho_3(\alpha) & = & \rho_3(\tau_1) \\ \rho(\beta) & = & \rho_3(\beta) & = & \rho_3(\tau_2) \end{array}$$

  Therefore, $\rho_1(\tau_1) = \rho_3(\tau_1)$ and $\rho_2(\tau_2) = \rho_3(\tau_2)$. Hence, $\rho_1(\sigma) = \rho_3(\sigma)$ and $\rho_2(\sigma) = \rho_3(\sigma)$, which imply $\rho_1(\sigma) = \rho_2(\sigma)$. Thus σ is forced to the same value.
- Assume σ is forced in $C_{i,j}$ because σ_1 is forced to a non-$\bot$ value and $\sigma_1 \Rightarrow \tau_1 \in C_{i,j}$ and is forced in $C_{i',j'}$ because σ_2 is forced to a non-$\bot$ value and $\sigma_2 \Rightarrow \tau_2 \in C_{i',j'}$. Because C is a closed system, we must have two interface variables or auxiliary variables α and β with both $\alpha \Rightarrow \tau_1$ and $\beta \Rightarrow \tau_2$ appearing in C. Since σ_1 and σ_2 are forced, then we must have $\rho(\alpha) = \rho^!(\sigma_1)$ and $\rho(\beta) = \rho^!(\sigma_2)$, thus σ must be forced to the same value by the previous case.

- Assume σ is forced in $C_{i,j}$ because $\rho(\alpha) \neq \perp$ and $\alpha \Rightarrow \tau_1 \in C_{i,j}$ and forced in $C_{i',j'}$ because σ_2 is forced to a non-$\perp$ value and $\sigma_2 \Rightarrow \tau_2 \in C_{i',j'}$. This case is similar to the previous case.

- The remaining case, where σ is forced in $C_{i,j}$ because σ_1 is forced to a non-$\perp$ value and $\sigma_1 \Rightarrow \tau_1 \in C_{i,j}$ and is forced in $C_{i',j'}$ because $\rho(\alpha) \neq \perp$ and $\alpha \Rightarrow \tau_2 \in C_{i',j'}$, is symmetric to the above case.

□

5.4.2 Entailment of pair constraints

In the previous subsection, we saw that a closed system can be decomposed into pairs of conditional constraints. In this section, we show how to efficiently determine entailment if the right-hand side consists of a pair of conditional constraints.

Lemma 5.4.5 Let C_1 be a system of conditional constraints and C_2 be a system of equality constraints with $E = \mathrm{fv}(C_1) \cap \mathrm{fv}(C_2)$. The decision problem $C_1 \models_E C_2$ is solvable in polynomial time.

Proof. Consider the following algorithm. We first solve C_1 using CondResolve, and add the terms appearing in C_2 to the resulting term graph for C_1. Then for any two terms appearing in the term graph, we decide, using the simple entailment algorithm in Figure 5.7, whether the two terms are the same. For terms which are equivalent we merge their equivalence classes. Next, for each of the constraints in C_2, we merge the left and right sides. For any two non-congruent classes that are unified, we require at least one of the representatives be a variable in $\mathrm{fv}(C_2) \setminus E$. If this requirement is not met, the entailment does not hold. Otherwise, the entailment holds.

If the requirement is met, then it is routine to verify that the entailment holds. Suppose the requirement is not met, *i.e.*, there exist two non-congruent classes which are unified and none of whose ecrs is a variable in $\mathrm{fv}(C_2) \setminus E$. Since the two classes are non-congruent, we can choose a satisfying valuation for C_1 which maps the two classes to different values. This must be possible because, otherwise, we would have proven that they are the same with the simple entailment algorithm for conditional constraints. The valuation $\rho\mid_E$ cannot be extended to a satisfying valuation for C_2 because, otherwise, this contradicts the fact that $C_1 \cup C_2$ entails the equivalence of the two non-congruent terms.

□

Theorem 5.4.6 Let C_1 be a system of conditional constraints. Let $C_=$ be a system of equality constraints. The following three decision problems can be solved in polynomial time:

1. $C_1 \models_E C_= \cup \{\alpha \Rightarrow \tau_1, \beta \Rightarrow \tau_2\}$, where $\alpha, \beta \in E$.
2. $C_1 \models_E C_= \cup \{\alpha \Rightarrow \tau_1, \mu \Rightarrow \tau_2\}$, where $\alpha \in E$ and $\mu \notin E$.
3. $C_1 \models_E C_= \cup \{\mu_1 \Rightarrow \tau_1, \mu_2 \Rightarrow \tau_2\}$, where $\mu_1, \mu_2 \notin E$.

Proof.

1. For the case $C_1 \models_E C_= \cup \{\alpha \Rightarrow \tau_1, \beta \Rightarrow \tau_2\}$, notice that $C_1 \models_E C_= \cup \{\alpha \Rightarrow \tau_1, \beta \Rightarrow \tau_2\}$ iff the following entailments hold

   - $C_1 \cup \{\alpha = \bot, \beta = \bot\} \models_E C_=$
   - $C_1 \cup \{\alpha = \bot, \beta = \nu_1 \to \nu_2\} \models_E C_= \cup \{\beta = \tau_2\}$
   - $C_1 \cup \{\alpha = \sigma_1 \to \sigma_2, \beta = \bot\} \models_E C_= \cup \{\alpha = \tau_1\}$
   - $C_1 \cup \{\alpha = \sigma_1 \to \sigma_2, \beta = \nu_1 \to \nu_2\} \models_E C_= \cup \{\alpha = \tau_1, \beta = \tau_2\}$

   where σ_1, σ_2, ν_1, and ν_2 are fresh variables not in $\mathrm{fv}(C_1) \cup \mathrm{fv}(C_2)$.

   Notice that each of the above entailments reduces to entailment of equality constraints, which can be decided in polynomial time by Lemma 5.4.5.

2. For the case $C_1 \models_E C_= \cup \{\alpha \Rightarrow \tau_1, \mu \Rightarrow \tau_2\}$, we consider two cases:

   - $C_1 \cup \{\alpha = \bot\} \models_E C_= \cup \{\mu \Rightarrow \tau_2\}$;
   - $C_1 \cup \{\alpha = \sigma_1 \to \sigma_2\} \models_E C_= \cup \{\alpha = \tau_1, \mu \Rightarrow \tau_2\}$

where σ_1 and σ_2 are fresh variables not in $\text{fv}(C_1) \cup \text{fv}(C_2)$.

We have a few cases:

- $\text{ECR}(\mu) = \perp$
- $\text{ECR}(\mu) = \tau_1 \to \tau_2$
- $\text{ECR}(\mu) \in E$
- $\text{ECR}(\mu) \notin E$

Notice that the only interesting case is the last case ($\text{ECR}(\mu) \notin E$) when there is a constraint $\beta = \tau$ in $C_=$ and μ appears in τ. For this case, we consider all the $\mathcal{O}(n)$ resulting entailments by setting β to some appropriate value according to the structure of τ, *i.e.*, we consider all the possible values for β. For example, if $\tau = (\mu \to \perp) \to \mu$, we consider the following cases:

- $\beta = \perp$;
- $\beta = \perp \to \nu_1$;
- $\beta = (\perp \to \nu_2) \to \nu_1$;
- $\beta = ((\nu_3 \to \nu_4) \to \nu_2) \to \nu_1$

where ν_1,ν_2,ν_3, and ν_4 are fresh variables.

Each of the entailments will have only equality constraints on the right-hand side. Thus, these can all be decided in polynomial time. Together, the entailments can be decided in polynomial time.

3. For the case $C_1 \models_E C_= \cup \{\mu_1 \Rightarrow \tau_1, \mu_2 \Rightarrow \tau_2\}$, we again apply the idea from the second case. The sub-case which is slightly different is, for example, when μ_2 appears in τ_1 only. In this case, for some β and τ, we have $\beta = \tau$ is in $C_=$ where μ_1 occurs in τ. Let $\tau' = \tau[\tau_1/\mu_1]$, where $\tau[\tau_1/\mu_1]$ denotes the type obtained from τ by replacing each occurrence of μ_1 by τ_1. Again, by assigning β appropriate values according to the structure of τ', we can consider $\mathcal{O}(n)$ entailments whose right-sides are equality constraint systems. Thus this form of entailment can also be decided in polynomial time.

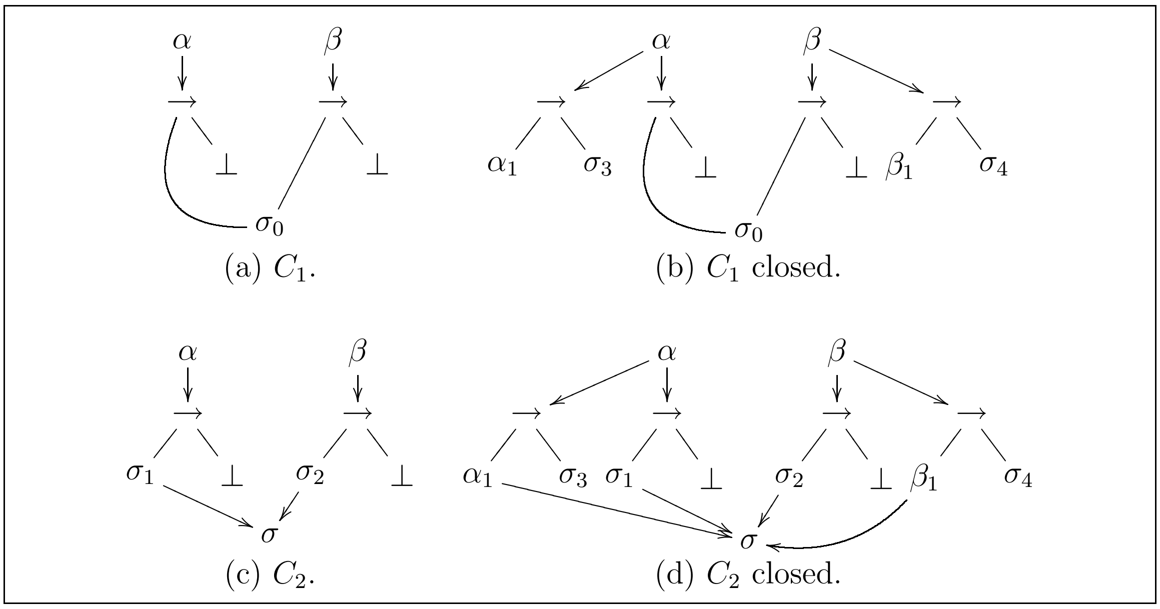


Figure 5.12: Example entailment.

□

5.4.3 Reduction of entailment to closed systems

We now reduce an entailment $C_1 \models_E C_2$ to entailment of closed systems, thus completing the construction of a polynomial time algorithm for restricted entailment over conditional constraints.

Unfortunately we cannot directly use the closed systems for C_1 and C_2 as demonstrated by the example in Figure 5.12. Figures 5.12a and 5.12c show two constraint systems C_1 and C_2. Suppose we want to decide $C_1 \models_{\{\alpha,\beta\}} C_2$. One can verify that the entailment does hold. Figures 5.12b and 5.12d show the closed systems for C_1 and C_2, which we name C_1' and C_2'. Note that we include the TR constraints of C_2 in C_1'. One can verify that the entailment $C_1' \models_{\{\alpha,\beta,\alpha_1,\beta_1\}} C_2'$ does not hold (take $\alpha = \beta = \perp$, $\alpha_1 = \perp \to \perp$, and $\beta_1 = \perp \to \top$, for example). The reason is that there is some information about α_1 and β_1 missing from C_1'. In particular, when both α_1 and β_1 are forced, we should have $\alpha_1 \Rightarrow \sigma'$ and $\beta_1 \Rightarrow \sigma'$ (actually in this case they satisfy the stronger relation that $\alpha_1 = \beta_1$). By replacing $\alpha \Rightarrow \alpha_1 \to \sigma_3$ and $\beta \Rightarrow \beta_1 \to \sigma_4$

with $\alpha = \alpha_1 \to \sigma_3$ and $\beta = \beta_1 \to \sigma_4$ (because that is when both are forced), we can decide that $\alpha_1 = \beta_1$. The following definition of a *completion* does exactly what we have described.

Definition 5.4.7 (Completion) Let C be a closed constraint system of C_0 w.r.t. E. Let A be the set of auxiliary variables. For each pair of variables α_i and β_j in A, let $C(\alpha_i, \beta_j) = C_{\mathrm{TR}}(\alpha_i) \cup C_{\mathrm{TR}}(\beta_j)$ (see Definition 5.4.1) and $C^{=}(\alpha_i, \beta_j)$ be the equality constraints by replacing $\Rightarrow$ with $=$ in $C(\alpha_i, \beta_j)$. Decide whether $C \cup C^{=}(\alpha_i, \beta_j) \vDash_{\{\alpha_i, \beta_j\}} \{\alpha_i \Rightarrow \sigma, \beta_j \Rightarrow \sigma\}$ (cf. Theorem 5.4.6). If the entailment holds, add the constraints $\alpha_i \Rightarrow \sigma_{(\alpha_i, \beta_j)}$ and $\beta_j \Rightarrow \sigma_{(\alpha_i, \beta_j)}$ to C, where $\sigma_{(\alpha_i, \beta_j)}$ is a fresh variable unique for α_i and β_j. The resulting constraint system is called the *completion* of C.

Theorem 5.4.8 Let C_1 and C_2 be two conditional constraint systems. Let C_2' be the closed system of C_2 w.r.t. to $E = \mathrm{fv}(C_1) \cap \mathrm{fv}(C_2)$ with A the set of auxiliary variables. Construct the closed system for C_1 w.r.t. E with A' the auxiliary variables, and add the TR constraints of closing C_2 to C_1 after closing C_1. Let C_1' be the completion of modified C_1. We have $C_1 \vDash_E C_2$ iff $C_1' \vDash_{E \cup A \cup A'} C_2'$.

Proof.

($\Leftarrow$): Assume $C_1' \vDash_{E \cup A \cup A'} C_2'$. Let $\rho \vDash C_1$. We can extend ρ to ρ' which satisfies C_1'. Since $C_1' \vDash_{E \cup A \cup A'} C_2'$, then there exists ρ'' such that $\rho'' \vDash C_2'$ with $\rho' |_{E \cup A \cup A'} = \rho'' |_{E \cup A \cup A'}$. Since $\rho'' \vDash C_2'$, we have $\rho'' \vDash C_2$. Also $\rho |_E = \rho' |_E = \rho'' |_E$. Therefore, $C_1 \vDash_E C_2$.

($\Rightarrow$): Assume $C_1 \vDash_E C_2$. Let $\rho \vDash C_1'$. Then $\rho \vDash C_1$. Thus there exists $\rho' \vDash C_2$ with $\rho |_E = \rho' |_E$. We extend $\rho' |_E$ to ρ'' with $\rho''(\alpha) = \rho'(\alpha)$ if $\alpha \in E$ and $\rho''(\alpha) = \rho(\alpha)$ if $\alpha \in (A \cup A')$. It suffices to show that ρ'' can be extended with mappings for variables in $\mathrm{fv}(C_2') \setminus (E \cup A \cup A') = \mathrm{fv}(C_2') \setminus (E \cup A)$, because $\rho'' |_{E \cup A \cup A'} = \rho |_{E \cup A \cup A'}$.

Notice that all the TR constraints in C_2' are satisfied by some extension of ρ'', because they also appear in C_1'. Also the constraints C_2 are satisfied by some extension of ρ''. It remains to show that the internal variables of C_2' are forced by ρ'' to the same value if they are forced by ρ'' in either the TR constraints or C_2. Suppose there is an internal variable σ forced to different values by ρ''. We can assume that σ is forced by ρ'' because $\rho''(\alpha_i) \neq \bot$ and $\alpha_i \Rightarrow \sigma$ and forced because $\rho''(\beta_j) \neq \bot$ and $\beta_j \Rightarrow \sigma$ for some interface or auxiliary variables α_i and β_j. Consider the interface variables $\textsc{root}(\alpha_i)$ and $\textsc{root}(\beta_j)$ (see Definition 5.4.1). Since the completion of C_1 does not include constraints $\{\alpha_i \Rightarrow \sigma', \beta_j \Rightarrow \sigma'\}$, thus we can assign $\textsc{root}(\alpha_i)$ and $\textsc{root}(\beta_j)$ appropriate values to force α_i and β_j to different non-$\bot$ values. However, C_2 requires α_i and β_j to have the same non-$\bot$ value. Thus, if there is an internal variable σ forced to different values by ρ'', we can construct a valuation which satisfies C_1, but the valuation restricted to E cannot be extended to a satisfying valuation for C_2. This contradicts the assumption that $C_1 \models_E C_2$. To finish the construction of a desired extension of ρ'' that satisfies C_2', we set the variables which are not forced to $\bot$.

One can easily verify that this valuation must satisfy C_2'. Hence $C_1' \models_{E \cup A \cup A'} C_2'$.

$\square$

5.4.4 Putting everything together

Theorem 5.4.9 Restricted entailment for conditional constraints can be decided in polynomial time.

Proof. Consider the problem $C_1 \models_E C_2$. By Theorem 5.4.8, it is equivalent to testing $C_1' \models_{E \cup A \cup A'} C_2'$ (see Theorem 5.4.8 for the appropriate definitions of C_1', C_2', A, and A'). Notice that C_1' and C_2' are constructed in polynomial time in the sizes of C_1 and C_2. Now by Theorem 5.4.4, this is equivalent to checking $\mathcal{O}(n^2)$ entailment problems of the form $C_1' \models_{E \cup A \cup A'} C_{2'_=} \cup \{c_i, c_j\}$, where $C_{2'_=}$ denote the equality constraints of C_2' and c_i and c_j are two conditional constraints of C_2'. And by Theorem 5.4.6, we can decide each of these entailments in polynomial time. Put everything together, we have a polynomial time algorithm for restricted entailment over conditional constraints.

$\square$

5.5 Extended conditional constraints

In this section, we consider a natural extension of the standard conditional constraint language. This section is helpful for a comparison between this constraint language with the standard conditional constraint language, which we consider in Section 5.3. The results in this section provide a clear boundary between tractable and intractable constraint languages in terms of entailment.

We extend the language with the following construct extending the conditional constraints used in previous sections. The new construct is

$$\alpha \Rightarrow (\tau_1 = \tau_2),$$

which holds iff either $\alpha = \bot$ or $\tau_1 = \tau_2$. We call this form of constraints *extended conditional equality constraints*.

To see that this construct indeed extends $\alpha \Rightarrow \tau$, notice that $\alpha \Rightarrow \tau$ can be encoded in the new constraint language as

$$\alpha \Rightarrow (\alpha = \tau).$$

This extension is interesting because many equality based program analyses can be naturally expressed with this form of constraints. An example analysis that uses this form of constraints is the equality based flow analysis for higher order functional languages [Pal98]. Additionally it can be used as a boundary for separating tractable and intractable constraint languages.

Note that satisfiability for this extension can still be decided in almost linear time with basically the same algorithm outlined for conditional equality constraints. We consider both simple entailment and restricted entailment for this extended language.

5.5.1 Simple entailment

Let C be a constraint system with $\alpha \Rightarrow (\beta = \gamma)$ a particular constraint c. We want to decide whether $C \vDash c$. Notice that $C \vDash c$ iff $C \cup \{\alpha = \top\} \vDash \beta = \gamma$ and $C \cup \{\alpha = \alpha_1 \to \alpha_2\} \vDash \beta = \gamma$, where α_1 and α_2 are fresh variables not appearing in C. Thus it suffices to consider entailment $C \vDash \alpha = \beta$.

Theorem 5.5.1 Let C be a extended constraint system, we can decide whether $C \models \alpha = \beta$ in polynomial time.

The algorithm is given in Figure 5.13. We give the basic idea of the algorithm. We consider the following cases:

- $\alpha = \bot$, $\beta = \top$;
- $\alpha = \bot$, $\beta = \beta_1 \to \beta_2$;
- $\alpha = \top$, $\beta = \bot$;
- $\alpha = \top$, $\beta = \beta_1 \to \beta_2$;
- $\alpha = \alpha_1 \to \alpha_2$, $\beta = \beta_1 \to \beta_2$.

For the first four cases, if adding any of the corresponding constraints to C makes the constraint system satisfiable, then $C \not\models \alpha = \beta$. For the last case where $\alpha = \alpha_1 \to \alpha_2$ and $\beta_1 \to \beta_2$, if the constraints make C unsatisfiable, then $C \models \alpha = \beta$. If the constraints make C satisfiable, then we recurse with $C \cup \{\alpha = \alpha_1 \to \alpha_2, \beta = \beta_1 \to \beta_2\} \models \alpha_1 = \beta_1$ and $C \cup \{\alpha = \alpha_1 \to \alpha_2, \beta = \beta_1 \to \beta_2\} \models \alpha_2 = \beta_2$. This naive algorithm may run in exponential time, we show, in the algorithm, how to use precomputed information to reduce the algorithm to polynomial time. The detailed algorithm is given in Figure 5.13. Step 8 of the algorithm is the key to reduce the running time from exponential to polynomial. The idea is if in Step 7, we know that $\alpha_1 = \beta_1$, then this information can be used in showing that $\alpha_2 = \beta_2$.

Proof.

The algorithm is obviously correct. We need to verify that the algorithm runs in polynomial time in the size of C. First notice that the depth that the algorithm recurses is at most $O(|C|)$ times. Since each time two conditional constraints are changed to equality constraints and there are at most $O(|C|)$ number of conditional constraints, the total number of recursive calls may be exponential. However, the second argument of each recursive call are two terms from the original term graph for C (except maybe the last call in which case the algorithm outputs NO). There

$\text{SE}(C, \alpha = \beta) = \text{YES}$ iff $C \vDash \alpha = \beta$.

1. Solve C. If $\text{ECR}(\alpha) = \text{ECR}(\beta)$, return YES, else continue.
2. If $C \cup \{\alpha = \bot, \beta = \top\}$ is satisfiable, return NO, else continue.
3. If $C \cup \{\alpha = \bot, \beta = \beta_1 \to \beta_2\}$ is satisfiable, return NO, else continue.
4. If $C \cup \{\alpha = \top, \beta = \bot\}$ is satisfiable, return NO, else continue.
5. If $C \cup \{\alpha = \top, \beta = \beta_1 \to \beta_2\}$ is satisfiable, return NO, else continue.
6. If $C \cup \{\alpha = \alpha_1 \to \alpha_2, \beta = \beta_1 \to \beta_2\}$ is unsatisfiable, return YES, else continue.
7. If $\text{SE}(C \cup \{\alpha = \alpha_1 \to \alpha_2, \beta = \beta_1 \to \beta_2\}, \alpha_1 = \beta_1)$ returns NO, then return NO, else continue.
8. If $\text{SE}(C \cup \{\alpha = \alpha_1 \to \alpha_2, \beta = \beta_1 \to \beta_2, \underline{\alpha_1 = \beta_1}\}, \alpha_2 = \beta_2)$ returns NO, then return NO, else return YES.

Figure 5.13: Algorithm for simple entailment over extended conditional equality constraints.

are $O(|C|^2)$ such term pairs. Consider the call $\text{SE}(C \cup \{\alpha = \alpha_1 \to \alpha_2, \beta = \beta_1 \to \beta_2\}, \alpha_1 = \beta_1)$, where $\text{ECR}(\alpha_1)$ and $\text{ECR}(\beta_1)$ are two terms from C. If it returns YES, then $\text{SE}(C \cup \{\alpha = \alpha_1 \to \alpha_2, \beta = \beta_1 \to \beta_2, \alpha_1 = \beta_1\}, \alpha_2 = \beta_2)$ can return YES immediately without repeating the computation if $\text{ECR}(\alpha_2) = \text{ECR}(\alpha_1)$ and $\text{ECR}(\beta_2) = \text{ECR}(\beta_1)$. With this observation, we conclude that the running time of the algorithm is polynomial in $|C|$. $\square$

5.5.2 Restricted entailment

In this subsection, we consider the restricted entailment for extended conditional constraints. We show that the decision problem $C_1 \vDash_E C_2$ for extended conditional constraints is coNP-complete.

We define the decision problem NENT as the problem of deciding whether $C_1 \not\models_E C_2$, where C_1 and C_2 are systems of extended conditional equality constraints and $E = \mathrm{fv}(C_1) \cap \mathrm{fv}(C_2)$.

Theorem 5.5.2 The decision problem NENT for extended conditional constraints is in NP.

Proof.

Let C_1 and C_2 be two extended constraint systems. Let $E = \mathrm{fv}(C_1) \cap \mathrm{fv}(C_2)$.

For each variable α in $\mathrm{fv}(C_1)$, we guess whether α is $\bot$, $\top$, or $\alpha_1 \to \alpha_2$ for some fresh variables α_1 and α_2. We add these constraints to C_1 to obtain C_1'. For each α in E, we add to C_2 the constraints $\alpha = \bot$, $\alpha = \top$, or $\alpha = \alpha_1 \to \alpha_2$ depending on what we guessed for C_1. Notice now that C_1' is a system of equality constraints. C_2', however, may still have some conditional constraints. This means that our guess for the variables E needs to be refined to get rid of these conditional constraints in C_2'. In C_2', for each conditional constraint with a fresh variable α_i (the generated variables) as antecedent, we guess the value of α_i and add the corresponding constraints to both C_1' and C_2'. This process is repeated until there are no more conditional constraints in C_2' with any fresh variables as antecedents. Since there are at most $\mathcal{O}(|C_2|)$ number of conditional constraints in C_2, thus we make at most $\mathcal{O}(|C_2|)$ number of guesses. Finally, conditional constraints with variables in $\mathrm{fv}(C_2)\backslash E$ as antecedents are discarded since these constraints do not affect the solutions of the constraints w.r.t. E.

Let C_1'' and C_2'' be the resulting constraint systems. Notice they are equality constraints. Thus at the end, we turn the problem into entailment over equality constraints, which we can decide in polynomial time. The guessing step takes time polynomial in $|C_1|$ and $|C_2|$. Thus NENT is in NP.

$\square$

Next we show that the problem NENT is hard for NP, thus an efficient algorithm is unlikely to exist for the problem. The reduction actually shows that with extended conditional constraints the atomic restricted entailment is coNP-hard.

Theorem 5.5.3 The decision problem NENT is NP-hard.

Proof.

We reduce 3-CNFSAT to NENT. As mentioned, the reduction shows that even the atomic restricted entailment over extended conditional constraints is coNP-complete, contrary to the case over conditional equality constraints, for which we give a polynomial time algorithm.

Let ψ be a boolean formula in 3-CNF form and let $\{x_1, x_2, \ldots, x_n\}$ and $\{c_1, c_2, \ldots, c_m\}$ be the boolean variables and clauses in ψ respectively. For each boolean variable x_i in ψ, we create two term variables α_{x_i} and $\alpha_{\overline{x_i}}$, which we use to decide the truth value of x_i. The value $\bot$ is treated as the boolean value false and any non-$\bot$ value is treated as the boolean value true.

Note, in a graph, a constraint of the form $\alpha \Rightarrow (\tau_1 = \tau_2)$ is represented by

$$\begin{tikzcd}
 & \alpha \arrow[d] & \\
 & {} \arrow[dl, dash] \arrow[dr, dash] & \\
\tau_1 & & \tau_2
\end{tikzcd}$$

First we need to ensure that a boolean variable takes on at most one truth value. We associate with each x_i constraints C_{x_i}, graphically represented as

$$\begin{tikzcd}
 & \alpha_{x_i} \arrow[d] & & \alpha_{\overline{x_i}} \arrow[d] & \\
 & {} \arrow[dl, dash] \arrow[dr, dash] & & {} \arrow[dl, dash] \arrow[dr, dash] & \\
\bot & & \tau_{x_i} & & \top
\end{tikzcd}$$

where τ_{x_i} is some internal variable. These constraints guarantee that at least one of α_{x_i} and $\alpha_{\overline{x_i}}$ is $\bot$. These constraints still allow both α_{x_i} and $\alpha_{\overline{x_i}}$ to be $\bot$, which we deal with below.

In the following, let $\alpha_{\overline{\overline{x}}} = \alpha_x$. For each clause $c_i = c_i^1 \vee c_i^2 \vee c_i^3$ of ψ, we create constraints C_{c_i} that ensure every clause is satisfied by a truth assignment. A clause is satisfied if at least one of the literals is true, which is the same as saying that the

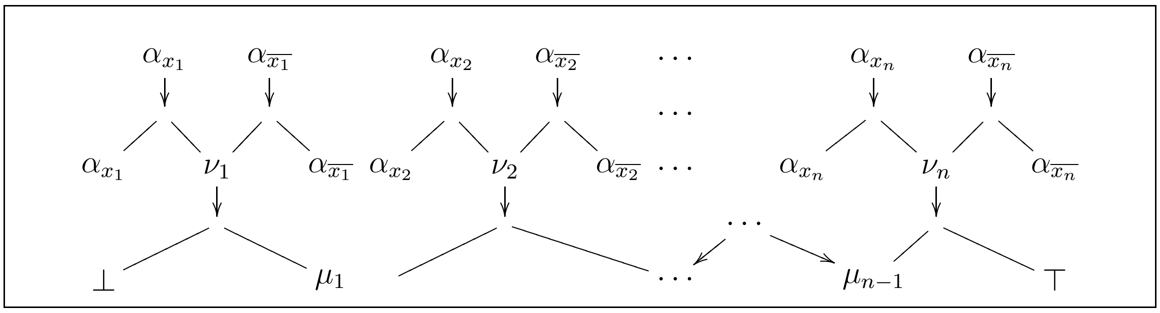
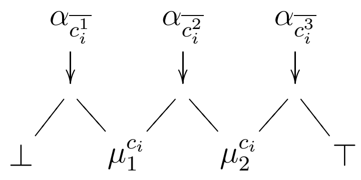


Figure 5.14: Constructed constraint system C_2.

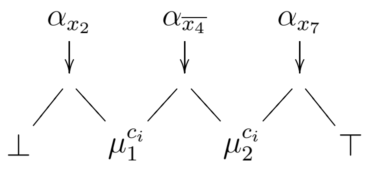
negations of the literals cannot all be true simultaneously. The constraints are



$\alpha_{\overline{c_i^1}}$
$\alpha_{\overline{c_i^2}}$
$\alpha_{\overline{c_i^3}}$
$\perp$
$\mu_1^{c_i}$
$\mu_2^{c_i}$
$\top$

where $\mu_1^{c_i}$ and $\mu_2^{c_i}$ are internal variables associated with c_i.

As an example consider $c_i = \overline{x_2} \vee x_4 \vee \overline{x_7}$. The constraints C_{c_i} are



We let C_1 be the union of all the constraints C_{x_i} and C_{c_j} for $1 \leq i \leq n$ and $1 \leq j \leq m$, *i.e.*,

$$C_1 = (\bigcup_{i=1}^{n} C_{x_i}) \cup (\bigcup_{j=1}^{m} C_{c_j})$$

There is one additional requirement that we want to enforce: not both α_{x_i} and $\alpha_{\overline{x_i}}$ are $\perp$. This cannot be enforced directly in C_1. We construct constraints for C_2 to enforce this requirement. The idea is that if for any x_i, the term variables α_{x_i} and $\alpha_{\overline{x_i}}$ are both $\perp$, then the entailment holds.

We now proceed to construct C_2. The constraints C_2 represented graphically are shown in Figure 5.14. In the constraints, all the variables except α_{x_i} and $\alpha_{\overline{x_i}}$ are internal variables. These constraints can be used to enforce the requirement that for all x_i at least one of α_{x_i} and $\alpha_{\overline{x_i}}$ is non-$\bot$. The intuition is that if α_{x_i} and $\alpha_{\overline{x_i}}$ are both $\bot$, the internal variable ν_i can be $\bot$, which breaks the chain of conditional dependencies along the bottom of Figure 5.14, allowing $\mu_1, \ldots, \mu_{i-1}$ to be set to $\bot$ and $\mu_i, \ldots, \mu_{n-1}$ to be set to $\top$.

We let the set of interface variables $E = \{\alpha_{x_i}, \alpha_{\overline{x_i}} \mid 1 \leq i \leq n\}$. We claim that ψ is satisfiable iff $C_1 \nvDash_E C_2$.

Claim 5.5.4 ψ is satisfiable iff $C_1 \nvDash_E C_2$.

Proof.

Assume that ψ is satisfiable. Let $f : \{x_i \mid 1 \leq i \leq n\} \to \{0, 1\}$ be a satisfying assignment for ψ. We construct from f a valuation ρ for the variables E such that ρ can be extended to a satisfying valuation for C_1 while it cannot be extended to a satisfying valuation for C_2. The existence of such a ρ is sufficient to conclude that $C_1 \nvDash_E C_2$. We construct ρ as follows

$$\begin{cases} \rho(\alpha_{x_i}) = \bot \ \wedge \ \rho(\alpha_{\overline{x_i}}) = \top & \text{if } f(x_i) = 0 \\ \rho(\alpha_{x_i}) = \top \ \wedge \ \rho(\alpha_{\overline{x_i}}) = \bot & \text{if } f(x_i) = 1 \end{cases}$$

The valuation ρ can be extended to a satisfying valuation ρ_1 for C_1. For each boolean variable x_i, there is a unique way to extend ρ to satisfy the constraints in C_{x_i}, namely $\rho_1(\tau_{x_i}) = \bot$ if $\rho(\alpha_{x_i}) = \top$ and $\rho_1(\tau_{x_i}) = \top$ otherwise. For each clause $c_i = c_i^1 \vee c_i^2 \vee c_i^3$, at least one of $f(c_i^1)$, $f(c_i^2)$, and $f(c_i^3)$ is 1. Thus, at least one of $\rho(\alpha_{\overline{c_i^1}})$, $\rho(\alpha_{\overline{c_i^2}})$, and $\rho(\alpha_{\overline{c_i^3}})$ is $\bot$. Assume $\rho(\alpha_{\overline{c_i^j}})$ is $\bot$ for some j with $1 \leq j \leq 3$. We map $\rho'(\mu_k^{c_i}) = \bot$ for all $1 \leq k < j$ and $\rho'(\mu_k^{c_i}) = \top$ for all $j \leq k < 3$. The extension clearly satisfies the constraints C_{c_i} for all c_i. Thus ρ can be extended to a satisfying valuation for C_1.

As for C_2, ρ cannot be extended to a satisfying assignment. Notice that it requires ν_i to be mapped to $\top$ for all i. This would require mapping μ_1 to $\bot$ and μ_{n-1} to $\top$ and mapping $\mu_i = \mu_j$ for all $1 \leq i, j \leq n-1$, which is impossible. Thus all extensions of ρ do not satisfy C_2. And therefore, we have $C_1 \nvDash_E C_2$.

For the other direction, assume that $C_1 \nvDash_E C_2$. Then there exists a $\rho_1 \vDash C_1$ and there does not exist a $\rho_2 \vDash C_2$ with $\rho_1(\alpha) = \rho_2(\alpha)$ for all $\alpha \in E$. Since C_1 is satisfiable, this is equivalent to saying that there exists a ρ on the variables E such that ρ can be extended to a satisfying valuation for C_1 and no extension of ρ satisfies C_2.

We construct from ρ a satisfying assignment for the boolean formula ψ. First notice that for each boolean variable x_i, ρ must map exactly one of the two type variables α_{x_i} and $\alpha_{\overline{x_i}}$ to $\perp$ and exactly one to a non-$\perp$ value. To see this notice $\rho(\alpha_{x_i})$ and $\rho(\alpha_{\overline{x_i}})$ cannot both be non-$\perp$, or the constraints C_{x_i} would then require $\perp$ and $\top$ to be unified. If $\rho(\alpha_{x_i})$ and $\rho(\alpha_{\overline{x_i}})$ are both $\perp$, then ρ can be extended to a satisfying valuation for C_2. In particular, $\rho'(\nu_i) = \perp$ if both $\rho(\alpha_{x_i})$ and $\rho(\alpha_{\overline{x_i}})$ are $\perp$. For the μ_j's, we let $\rho'(\mu_j) = \perp$ if $j < i$ and $\rho(\mu_j) = \top$ if $j \geq i$. It is easy to see that $\rho' \vDash C_2$. Thus we have shown for each variable x_i exactly one of $\rho(\alpha_{x_i})$ and $\rho(\alpha_{\overline{x_i}})$ is $\perp$ and exactly one is a non-$\perp$ value.

Now we can show how to construct from ρ a satisfying assignment f of ψ. We let

$$\begin{cases} f(x_i) = 0 \ \wedge \ f(\overline{x_i}) = 1 & \text{if } \rho(\alpha_{x_i}) = \perp \\ f(x_i) = 1 \ \wedge \ f(\overline{x_i}) = 0 & \text{otherwise} \end{cases}$$

We show that f satisfies each clause of ψ. Let $c_i = c_i^1 \vee c_i^2 \vee c_i^3$ be a clause of ψ. Consider the constraints C_{c_i}. At least one of $\rho(\alpha_{\overline{c_i^1}})$, $\rho(\alpha_{\overline{c_i^2}})$, and $\rho(\alpha_{\overline{c_i^3}})$ must be $\perp$. W.L.O.G., assume that $\rho(\alpha_{\overline{c_i^1}})$ is $\perp$. Then $\rho(\alpha_{c_i^1})$ is non-$\perp$. Thus, $f(c_i^1)$ is 1. Therefore, f satisfies the clause c_i. Hence, f satisfies every clause of ψ, and f satisfies ψ itself.

□

To prove the NP-hardness result, observe that the described reduction is a polynomial-time reduction. Thus, the decision problem NENT is NP-hard.

□

We thus have shown that the entailment problem over extended conditional constraints is coNP-complete. The result holds even if all the constraints are restricted to be atomic.

Theorem 5.5.5 The decision problem $C_1 \vDash_E C_2$ over extended conditional constraints is coNP-complete.

5.6 Related issues

We have given a complete characterization of the complexities of deciding entailment for conditional equality constraints and extended conditional constraints. There are a few related problems to be considered:

- What happens if we allow recursive types?
- What is the relationship with strict constructors (*i.e.*, if $c(\bot) = \bot$)?
- What is the relationship with a type system equivalent to the equality-based flow systems [Pal98]? In this type system, the only subtype relation is given by $\bot \leq t_1 \rightarrow t_2 \leq \top$, and there is no non-trivial subtyping between function types.

We believe the same or similar techniques can be used to address the above mentioned problems, and many of the results should carry over to these problem domains. These problems are left for future research.

Chapter 6

Related Work

The inspiration for this thesis is the need for scalable and expressive program analysis and type systems. In earlier work, we investigated heuristic methods for simplifying constraints. From this initial work, following the need of more powerful simplification techniques and enabling expressive analysis techniques, it became interesting to systematically understand the precise complexity and related problems of constraint simplification, and thus entailment. In the discussion of related work, we attempt to separate results as either focusing primarily on complexity analysis or practical heuristics.

6.1 Theoretical development

6.1.1 Work on type simplification

A few researchers consider the semantic notions for subtyping constraint simplification. The most powerful one is the notion of *observational equivalence* defined in [TS96]: two sets of constraints are observationally equivalent if replacing one with the other does not affect the results of an analysis. This corresponds to the notion of existential entailment and subtyping constrained types. A similar notion is used in [Pot96] for simplifying subtyping constraints.

Aiken, Wimmers, and Palsberg [AWP97] consider the problem of representing polymorphic types. They identify a simple algorithm and show it sound and complete for a few simple type languages. The *optimal type* of a polymorphic type is defined as a type which is equivalent and contains least number of type variables. The goal of this work, as put by the authors, is to understand why it seems difficult to get a practical system combining polymorphism and subtyping. They leave open the problem of optimal representation for polymorphically constrained types.

6.1.2 Work on entailment

Henglein and Rehof consider the problem of subtyping constraint entailment of the form $C \vDash \alpha \leq \beta$ [HR97, HR98]. The types are constructed from a finite lattice of base elements with the function ($\rightarrow$) and product ($\times$) constructors. They consider four cases for this problem.

- *structural subtyping over finite (simple) types*
  The entailment problem $C \vDash \alpha \leq \beta$ is shown to be coNP-complete [HR97].
- *structural subtyping over recursive types*
  The problem is shown to be PSPACE-complete [HR98].
- *non-structural subtyping over finite (simple) types*
  The problem is shown to be PSPACE-hard [HR98].
- *non-structural subtyping over recursive types*
  The problem is shown to be PSPACE-hard [HR98].

Niehren and Priesnitz also consider the problem of non-structural subtype entailment. They show that a natural subproblem is PSPACE-complete [NP99] and characterize non-structural subtype entailment over the signature $\{f(,), \perp, \top\}$ with so-called P-automata [NP01]. They leave open the decidability of non-structural subtype entailment for this particular signature. In addition, it is not known whether this approach can be extended to work on arbitrary signatures.

6.1.3 Work on entailment in related domains

Niehren *et al.* consider the entailment problem of *atomic set constraints*, a restricted class of set constraints without union and intersections and interpreted over the Herbrand universe. They show entailment of the form $C \models \alpha \subseteq \beta$ is PSPACE-complete for atomic set constraints [NMT99].

Flanagan and Felleisen [FF97] consider the problem of simplifying a restricted class of set constraints. They study the problem in the context of the program analysis tool for Scheme MrSpidy, a tool for inferring runtime values of variables for static debugging. They study the problem of existential entailment and show that existential entailment problem is decidable (in exponential time and space) by reducing the problem to an extended version of regular tree grammar containment [CDG$^+$99, GS84]. They show the problem is PSPACE-hard by a polynomial time reduction from nondeterministic finite state automata containment (which is PSPACE-complete) to the set constraint entailment problem. An exact characterization of the complexity of the problem remains open. The precise relationship of their entailment problems to those of atomic set constraints and subtyping constraints is not clear.

There is also related work in term rewriting and constraint solving over trees in general [CT94, Com90]. Part of the work in this thesis is inspired by work in this area. Maher shows the first-order theory of finite trees, infinite trees, and rational trees is decidable by giving a complete axiomatization [Mah88]. Many researchers consider various order relations among trees, similar to the subtype orders. Venkataraman studies the first-order theory of sub-term ordering over finite trees. The existential fragment is shown to be NP-complete and the $\exists\forall$-fragment to be undecidable [Ven87]. Müller *et al.* study the first order theory of feature trees and show it undecidable [MNT01]. Comon and Treinen show the first-order theory of lexicographic path ordering is undecidable [CT97]. Automata-theoretic constructions are used to obtain decidability results for many theories. Büchi uses finite word automata to show the decidability of WS1S and S1S [Büc60]. Finite automata are also used to construct alternative proofs of decidability of Presburger arithmetic [WB95, BC96], and Rabin's decidability proofs of WS2S and S2S are based on tree automata [Rab69].

6.1.4 Scalable type-based program analysis

Motivated by understanding the principles for building scalable program analysis, this thesis is related to the large body of work in this area [Das00,FRD00,Ste96,RF01, SFA00, FFSA98, AFFS98, HM97a, EST95, TS96, Pot01, Pot96, MW97, FA96, FFK$^+$96, FF97, Rep00, MR97, HM97b, Hei92, Hei94]. From a theoretical point of view, two pieces of work stand out, which we mention in detail. Recent work by Rehof and Fähndrich [RF01] extends the idea of Reps and Horowitz on interprocedural dataflow analysis [RHS95] to polymorphic label-flow in type-based program analysis [Mos96]. It explores the idea of instantiating the constraints implicitly through so-called instantiation constraints, rather than duplicating the complete constraint set as done in [Mos96] and most other cases. This technique crucially relies upon on a fixed type structure (pre-computed or provided to the analysis), and it is orthogonal to the problems studied in this thesis.

The other work is by Reps [Rep00] on the decidability of context-sensitive value flow analysis, which encompasses many common flow analysis such as set-based analysis [Hei92] and pointer analysis [And94]. It is shown that the general problem is undecidable by a reduction from a variant of PCP [Pos46].

6.2 Practical heuristics

There is also much research on practical heuristics for constraint simplification.

6.2.1 Type and constraint simplification

Fähndrich and Aiken identify a few simple techniques for simplifying polymorphically constrained types and demonstrate better scalability [FA96]. Pottier provides a sound but incomplete algorithm for simplifying polymorphically constrained types and shows some improvement [Pot96]. Marlow and Wadler designed and implemented a subtyping system for Erlang [MW97]. They give a sound approximate algorithm for deciding entailment and claim the algorithm to be complete. They have implemented a prototype system which shows some promise of being practical. Flanagan and Felleisen identify a few practical techniques for simplifying a form of set constraints [FF97]. They show promising reduction in both constraint size and analysis time.

6.2.2 Scalable constraint resolution techniques

Efficient constraint resolution algorithms and implementations can also be viewed as simplification techniques because we are able to handle larger sets of constraints. Therefore, we also look at some work on efficient constraint resolution.

Much progress has been made in recent years on solving inclusion constraints, including subtyping constraints and set constraints. In [FFSA98], Fähndrich *et al.* introduce the technique of online cycle elimination, which provides promising performance improvement. It is a simple yet effective technique for eliminating constraint cycles (chains of inclusion $X \subseteq Y \subseteq Z \cdots \subseteq X$). The technique is validated using a cubic time pointer analysis for C [And94]. In a following paper, Su *et al.* suggest *projection merging*, another technique to use in combination with online cycle elimination. The idea is to merge many upper bounds on a variable into a single upper bound. Using cycle elimination and projection merging together results in an analysis that can perform pointer analysis on C programs with half million source lines [SFA00].[1] Experiments have also shown that cycle elimination and projection merging help dramatically with the scaling of a version of polymorphic pointer analysis based on inclusion constraints [FFA00].

In a recent work [HT01], Heintze also considers the problem of solving inclusion constraints efficiently. In particular, it provides a new algorithm for implementing dynamic transitive closure, which is at the core of solving inclusion constraints. The idea is not to maintain the graph transitively closed. When information about a node is requested, a reachability step is used to obtain the result.[2] With further implementation techniques, this reachability computation can be done efficiently. As

[1]In [SFA00], we report an analysis time of approximately half an hour. Our original analysis was implemented in SML/NJ [MTH90], and with a port of the system to C, the analysis time drops to about two minutes.

[2]This is done similarly in [FFSA98, SFA00], a further reachability step is required at the end to compute the analysis information.

reported in [HT01], the analysis time for the same half million line C program as in [SFA00] is about 20 seconds.

Chapter 7

Conclusion and Future Work

In this chapter, we give a summary of the thesis, then discuss some open problems and issues, and possible directions for solving these problems.

7.1 Thesis summary

In this thesis, we have studied and addressed issues in scalable constraint and type-based program analysis. In particular, we have shown that the first-order theory of subtyping constraints is undecidable. The result may indicate that many of the open problems on entailment over non-structural subtyping might in fact be undecidable. The result is robust in the sense that it holds for any type language with a bottom element (or a top element) and at least one binary type constructor.

We have also shown that the monadic fragment of the first-order theory is decidable via an automata-theoretic reduction. By introducing constrained tree automata, we have shown a reduction of subtyping entailment over an arbitrary type signature to the emptiness problem for constrained tree automata. This automata-theoretic approach may provide a new attack on these difficult problems. Finally, we have shown polynomial-time algorithms for entailment and existential entailment over the domain of conditional equality constraints, which provides a weak form of non-structural subtyping. This work deepens our understanding of the power and complexity of subtyping as a means of expressing a type system or a program analysis. The conclusion we draw from the results is that even the simplest type structure $\{\bot, f\}$ has entailment problems that are quite subtle.

7.2 Open problems

There are many outstanding open problems in this area. We discuss the most important ones and provide a personal view on how these problems can be attacked. The most important and obvious open problems are the decidability and exact complexity of these entailment problems. We discuss them separately in detail.

7.2.1 Non-structural subtype entailment

Our personal view is that this problem is decidable and can be decided in PSPACE. There are two promising approaches to solving this problem. For the restricted signature $\{\bot, \top, f\}$ considered by Niehren and Priesnitz [NP99, NP01], the reduction to the universality problem over extended word automata seems promising.

For general signatures, there are two existing approaches. One is attempting to extend the approach of Niehren and Priesnitz to work on arbitrary signatures. The other is the approach outlined in this thesis, by considering the emptiness problem of constrained tree automata. Techniques and intuitions gained in solving the single non-trivial constructor case should be helpful in solving the general case.

7.2.2 Existential entailment and subtyping constrained types

For existential entailment and subtyping constrained types, virtually nothing is known at this point, except the lower bounds carried over from entailment. In particular, existential entailment in the case of finite structural subtyping is coNP-hard, and in all the other cases PSPACE-hard.

For structural subtyping, existential entailment seems relatively easy, and likely to be decidable. The procedure for deciding entailment might be extended to work for existential entailment as well, through some form of quantifier elimination [Mah88]. For non-structural subtyping, the problem seems much more subtle and difficult.

In fact, there is evidence in the study of the first-order theory of non-structural subtyping that this problem may in fact be undecidable. For example, we can adapt the undecidability proof for the emptiness problem of constrained automata to get an alternative proof of the undecidability of the first-order theory of non-structural subtyping. In this alternative proof, it appears that we construct a formula with only a single quantifier alternation, thus in the same fragment as the one that contains both existential entailment and subtyping constrained types.

7.2.3 The first-order theory of structural subtyping

We have shown that the first-order theory of non-structural subtyping is undecidable. However, the proof is not readily extended to the first-order theory of structural subtyping, which we leave as an interesting open problem. Because of the fundamental difference between structural and non-structural subtyping, and closer relationship between structural types and equality-based tree theories, it is reasonable to conjecture that the first-order theory of structural subtyping is decidable. We suspect the idea of quantifier elimination used in [Mah88] to prove that the first-order theory of trees is decidable can be extended to handle the first-order theory of structural subtyping.

Bibliography

[AC91] R. Amadio and L. Cardelli. Subtyping recursive types. In *Proceedings of the 18th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 104–118, Orlando, FL, USA, January 1991.

[AC93] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[AFFS98] A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proceedings of the Second International Workshop on Types in Compilation (TIC'98)*, pages 78–96, March 1998.

[AHU68] A. Aho, J. Hopcroft, and J. Ullmann. Time and tape complexity of pushdown automaton languages. *Information and Control*, 13(3):186–206, 1968.

[Aik99] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2–3):79–111, November 1999.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[Apo74] T. Apostol. *Mathematical Analysis.* Addison-Wesley, Reading, Massachusetts, 1974.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, 1986.

[AW92] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 329–340, June 1992.

[AW93] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[AWL94] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Proceedings of the 21th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 163–173, January 1994.

[AWP97] A. Aiken, E. Wimmers, and J. Palsberg. Optimal representations of polymorphic types with subtyping. In *Proceedings of 3rd International Symposium on Theoretical Aspects of Computer Software (TACS'97)*, pages 47–76, 1997.

[Bar91] H.P. Barendregt. *The Lambda Calculus.* Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, revised edition, 1991.

[BC96] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In *Proceedings of Trees in Algebra and Programming (CAAP'96)*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer-Verlag, 1996.

[Büc60] J. Büchi. Weak second order logic and finite automata. *Z. Math. Logik, Grundlag. Math.*, 5:66–62, 1960.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedual data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, 1988.

[Car96] L. Cardelli. Type systems. In A.B. Tucker, editor, *Handbook of Computer Science and Engineering*. CRC Press, 1996.

[CDG$^+$99] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1999.

[Col82] A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, London, 1982.

[Col84] A. Colmerauer. Equations and inequations on finite and infinite trees. In *2nd International Conference on Fifth Generation Computer Systems*, pages 85–99, 1984.

[Com90] H. Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4):387–411, 1990.

[CT94] H. Comon and R. Treinen. Ordering constraints on trees. In S. Tison, editor, *Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, pages 1–14, Edinburgh, Scotland, 11-13 April 1994. Springer Verlag.

[CT97] H. Comon and R. Treinen. The first-order theory of lexicographic path orderings is undecidable. *Theoretical Computer Science*, 176:67–87, April 1997.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):480–521, December 1985.

[Das00] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, N.Y., June 2000.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[EHM$^+$99] P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H. Sørensen, and M. Tofte. AnnoDomini: from type theory to Year 2000 conversion tool. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, New York, NY, USA, 1999.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings of the 10th annual Conference on Object-Oriented Programming Systems, Languages and Applications*, 1995.

[FA96] M. Fähndrich and A. Aiken. Making set-constraint based program analyses scale. In *First Workshop on Set Constraints at CP'96*, Cambridge, MA, August 1996. Available as Technical Report CSD-TR-96-917, University of California at Berkeley.

[FF97] C. Flanagan and M. Felleisen. Componential set-based analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[FFA00] J. Foster, M. Fähndrich, and A. Aiken. Monomorphic versus polymorphic flow-insensitive points-to analysis for c. In *Proceedings of the 7th International Static Analysis Symposium*, pages 175–198, 2000.

[FFK$^+$96] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[FFSA98] M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Montreal, CA, June 1998.

[FM89] Y. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In J. Díaz and F. Orejas, editors, *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages (CCIPL)*, volume 352 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1989.

[FM90] Y. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, June 1990.

[FRD00] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–263, N.Y., June 2000.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[GS84] F. Gécseg and M. Steinby. *Tree Automata.* Akademiai Kiado, 1984.

[Hei92] N. Heintze. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, 1992.

[Hei94] N. Heintze. Set based analysis of ML programs. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 306–17, June 1994.

[Hen88] F. Henglein. Type inference and semi-unification. In R. Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming (LFP)*, pages 184–197, Snowbird, Utah, July 1988.

[Hen92] F. Henglein. Global tagging optimization by type inference. In *1992 ACM Conference on Lisp and Functional Programming*, pages 205–215, June 1992.

[Hin97] J.R. Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, UK, 1997.

[HM97a] N. Heintze and D. McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–272, 1997.

[HM97b] N. Heintze and D. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings of the 1997 IEEE 12th Annual Symposium on Logic in Computer Science*, pages 342–351, 1997.

[HR97] F. Henglein and J. Rehof. The complexity of subtype entailment for simple types. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 352–361, 1997.

[HR98] F. Henglein and J. Rehof. Constraint automata and the complexity of recursive subtype entailment. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 616–627, 1998.

[HRB88] S. Horwitz, T. Reps, and D. Brinkley. Interprocedural slicing using dependence graphs. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, July 1988.

[HT01] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, N.Y., June 2001.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[JM79] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Proceedings of the 6th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 244–256, January 1979.

[JM94] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–582, May 1994.

[KPS93] D. Kozen, J. Palsberg, and M.I. Schwartzbach. Efficient recursive subtyping. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 419–428, 1993.

[KPS94] D. Kozen, J. Palsberg, and M.I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences (JCSS)*, 49(2):306–324, 1994.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 2 edition, 1987.

[Mah88] M.J. Maher. Complete axiomatizations of the algebras of the finite, rational and infinite trees. In *Proceedings of the Third IEEE Symposium on Logic in Computer Science*, pages 348–357, Edinburgh, UK, 1988. Computer Society Press.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Mit84] J.C. Mitchell. Coercion and type inference. In *11th Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, January 1984.

[Mit91] J.C. Mitchell. Type inference with simple types. *Journal of Functional Programming*, 1(3):245–285, 1991.

[Mit96] J.C. Mitchell. *Foundations of Programming Languages.* MIT Press, 1996.

[MNT01] M. Müller, J. Niehren, and R. Treinen. The first-order theory of ordering constraints over feature trees. *Discrete Mathematics and Theoretical Computer Science*, 4(2):193–234, September 2001.

[Mos96] C. Mossin. *Flow Analysis of Typed Higher-Order Programs.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[MPS86] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1-2):95–130, 1986.

[MR97] D. Melski and T. Reps. Interconvertibility of set constraints and context-free language reachability. In *Proceedings of the 1997 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, pages 74–89, June 1997.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[MW97] S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of the International Conference on Functional Programming (ICFP '97)*, pages 136–149, June 1997.

[NMT99] J. Niehren, M. Müller, and J. Talbot. Entailment of atomic set constraints is pspace-complete. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 285–294, 1999.

[NNH99] F. Nielson, H.R. Nielson, and C.L. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[NO80] C.G. Nelson and D.C. Oppen. Fast decision algorithm based on congruence closure. *JACM*, 1(2):356–364, 1980.

[NP99] J. Niehren and T. Priesnitz. Entailment of non-structural subtype constraints. In *Asian Computing Science Conference*, number 1742 in LNCS, pages 251–265. Springer, December 1999.

[NP01] J. Niehren and T. Priesnitz. Non-structural subtype entailment in automata theory. In *Proceedings of 4th International Symposium on Theoretical Aspects of Computer Software (TACS'01)*, number 2215 in LNCS, pages 360–384. Springer, 2001.

[OJ97] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 338–348, 1997.

[OW92] P. O'Keefe and M. Wand. Type inference for partial types is decidable. In Bernd Krieg-Brückner, editor, *ESOP'92, 4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 408–417, Rennes, France, 26–28 February 1992. Springer.

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, January 1997.

[Pal98] J. Palsberg. Equality-based flow analysis versus recursive types. *ACM Transactions on Programming Languages and Systems*, 20(6):1251–1264, 1998.

[Pie02] B.C. Pierce. *Types and Programming Languages.* The MIT Press, Cambridge, MA, 2002.

[PL99] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 276–290, New York, NY, USA, 1999.

[PO95a] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, 1995.

[PO95b] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995.

[Pos46] E.L. Post. A variant of a recursively unsolvable problem. *Bull. of the Am. Math. Soc.*, 52, 1946.

[Pot96] F. Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 122–133, May 1996.

[Pot01] F. Pottier. Simplifying subtyping constraints: a theory. *Information & Computation*, 170(2):153–183, November 2001.

[PS91] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proceedings of the ACM Conference on Object-Oriented programming: Systems, Languages, and Applications*, pages 146–161, October 1991.

[PT96] V. Pratt and J. Tiuryn. Satisfiability of inequalities in a poset. *Fundamenta Informatica*, 28, 1996.

[PW78] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and Systems Sciences*, 16(2):158–167, 1978.

[PWO97] J. Palsberg, M. Wand, and P. O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9(1):49–67, 1997.

[Rab69] M.O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[Reh98] J. Rehof. *The Complexity of Simple Subtyping Systems*. PhD thesis, DIKU, 1998.

[Rep98] T. Reps. Program analysis via graph reachability. Technical Report CS-TR-1998-1386, University of Wisconsin, Madison, August 1998.

[Rep00] T. Reps. Undecidability of context-sensitive data-independence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162–186, January 2000.

[Rey69] J. C. Reynolds. *Automatic Computation of Data Set Definitions*, pages 456–461. Information Processing 68. North-Holland, 1969.

[RF01] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, London, United Kingdom, January 2001.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, January 1995.

[RM99] J. Rehof and T. Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2–3):191–221, November 1999.

[Rob71] J.A. Robinson. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.

[SA01] Z. Su and A. Aiken. Entailment with conditional equality constraints. In *Proceedings of European Symposium on Programming*, pages 170–189, April 2001.

[SAN$^+$02] Z. Su, A. Aiken, J. Niehren, T. Priesnitz, and R. Treinen. The first-order theory of subtyping constraints. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 203–216, 2002.

[SFA00] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–95, 2000.

[Shi88] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[ST94] G. Smolka and R. Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, 1994.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[Str95] B. Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, MA, USA, second, reprinted with corrections august, 1995 edition, 1995.

[Tar75] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *JACM*, pages 215–225, 1975.

[Tha94] S.R. Thatte. Type inference with partial types. *Theoretical Computer Science*, 124(1):127–148, 1994.

[Tho90] W. Thomas. *Handbook of Theoretical Computer Science*, volume B, chapter Automata on Infinite Objects, pages 133–192. Elsevier, 1990.

[Tho96] W. Thomas. Languages, automata, and logic. Technical Report Rep. 9607, Inst. f. Informatik u. Prakt. Math., University of Kiel, 1996.

[Tiu92] J. Tiuryn. Subtype inequalities. In Andre Scedrov, editor, *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 308–317, Santa Cruz, CA, June 1992. IEEE Computer Society Press.

[Tre92] R. Treinen. A new method for undecidability proofs of first order theories. *Journal of Symbolic Computation*, 14(5):437–457, November 1992.

[Tre00] R. Treinen. Predicate logic and tree automata with tests. In J. Tiuryn, editor, *Foundations of Software Science and Computation Structures*, volume LNCS 1784, pages 329–343. Springer, 2000.

[Tre01] R. Treinen. Undecidability of the emptiness problem of reduction automata with component-wise tests. Available at `http://www.lsv.ens-cachan.fr/~treinen/publications.html`, June 2001.

[TS96] V. Trifonov and S. Smith. Subtyping constrained types. In *Proceedings of the 3rd International Static Analysis Symposium*, pages 349–365, September 1996.

[TW93] J. Tiuryn and M. Wand. Type reconstruction with recursive types and atomic subtyping. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development, 4th International Joint Conference CAAP/FASE*, LNCS 668, pages 686–701, Orsay, France, April 13–17, 1993. Springer-Verlag.

[Ven87] K.N. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *Journal of the ACM*, 34(2):492–510, April 1987.

[WB95] P. Wolper and B. Boigelot. An automata-theoretic approach to presburger arithmetic constraints. In *Static Analysis Symposium*, pages 21–32, 1995.

[Yel93] D.M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30(4):369–384, July 1993.