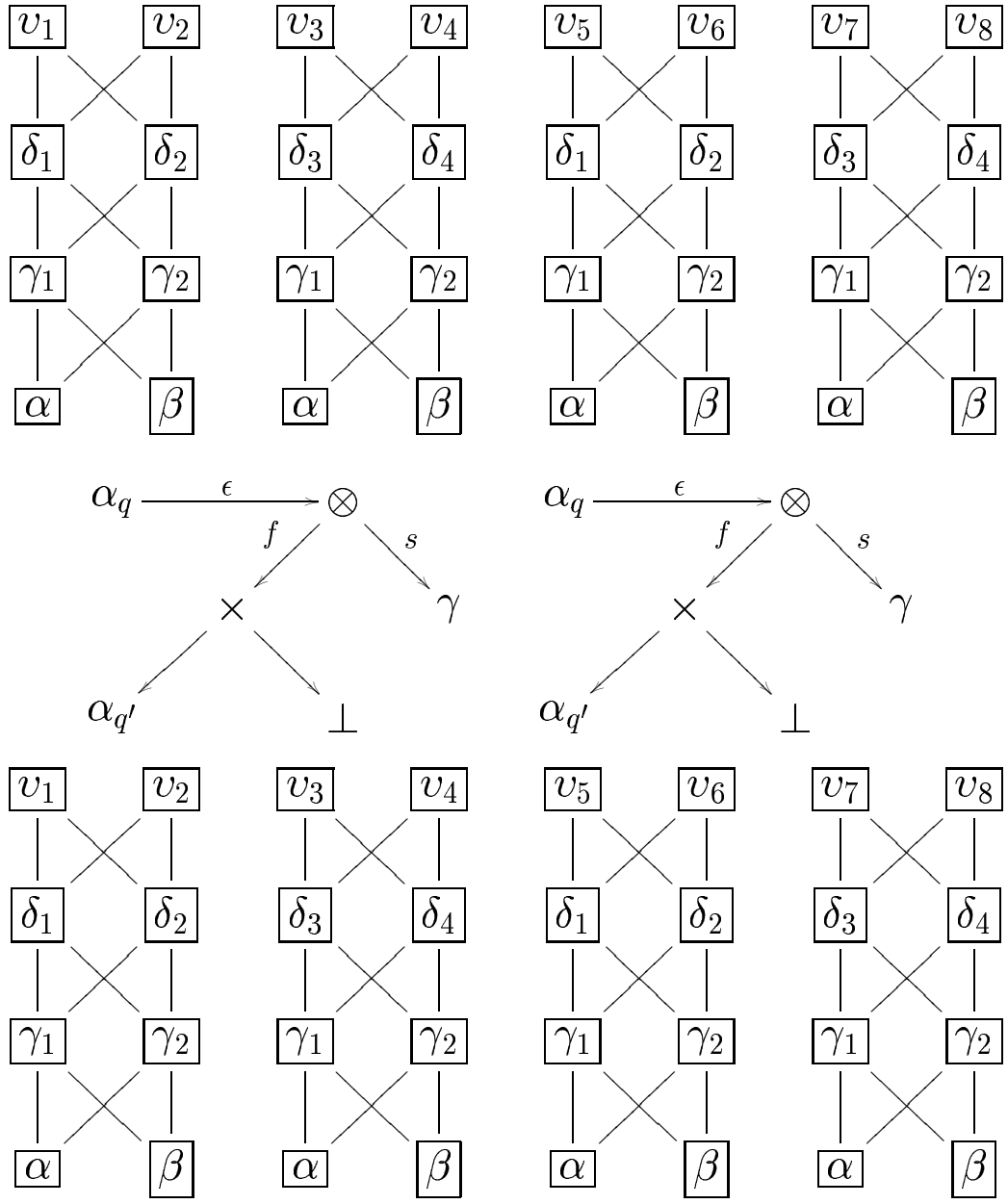
The Complexity of Simple Subtyping Systems

Ph.D. thesis

Jakob Rehof

DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen Ø, Denmark
Electronic mail: rehof@diku.dk
Phone : (+ 45) 35 32 14 08
Fax : (+ 45) 35 32 14 01


υ_1 υ_2 υ_3 υ_4 υ_5 υ_6 υ_7 υ_8
δ_1 δ_2 δ_3 δ_4 δ_1 δ_2 δ_3 δ_4
γ_1 γ_2 γ_1 γ_2 γ_1 γ_2 γ_1 γ_2
α β α β α β α β
α_q ϵ $\otimes$ f s $\times$ γ $\alpha_{q'}$ $\perp$
α_q ϵ $\otimes$ f s $\times$ γ $\alpha_{q'}$ $\perp$
υ_1 υ_2 υ_3 υ_4 υ_5 υ_6 υ_7 υ_8
δ_1 δ_2 δ_3 δ_4 δ_1 δ_2 δ_3 δ_4
γ_1 γ_2 γ_1 γ_2 γ_1 γ_2 γ_1 γ_2
α β α β α β α β

THESIS ADVISOR :

Fritz Henglein, DIKU, Copenhagen

THESIS COMMITTEE:

Dexter Kozen, Cornell, USA

Michael Schwartzbach, Aarhus, Denmark

Jerzy Tiuryn, Warsaw, Poland

Abstract

The main objective of this thesis is to understand why polymorphic subtype inference systems may be inherently difficult to scale up. This is done through a study of the complexity of *type size*, constraint *simplification* and constraint *entailment* in simple subtyping systems.

Simplification aims at presenting subtyping judgements in irredundant and succinct form. It is a bottleneck problem for scalable polymorphic subtype inference. Deciding entailment is a key problem in simplification and in information extraction from subtyping judgements.

We study the structure of principal typings and the limits of simplification in a variety of simple subtyping frameworks. Comparing the relative power of simplification in systems of increasing strength, we show that all known frameworks for atomic subtyping lead to worst case exponential dag–size of principal typings, thereby indicating that presenting typings succinctly is intrinsically harder than deciding typability, even in very well–behaved systems.

We study the complexity of deciding subtype entailment over lattices of type constants, for all combinations of atomic, structural, non–structural, finite and recursive subtype orders. Entailment turns out to be much more complex than satisfiability, which is in PTIME for all structures considered. We find that, while atomic entailment is linear time decidable, the addition of syntactic structure alone renders the entailment problem intractable. We show that entailment for finite structural subtyping is coNP–complete and PSPACE–complete for structural recursive subtyping, thereby indicating that recursive types incur additional computational complexity. Entailment for finite and recursive non–structural subtyping is shown to be PSPACE–hard, thereby indicating that, in the finite case, non–structural subtype orders are more complicated than structural ones.

Preface

This is a slightly revised version of a thesis submitted for the degree of Ph.D. at DIKU, Department of Computer Science at the University of Copenhagen, Denmark. It reports work done between October 1995 and March 1998 at DIKU, with a visit to Stanford University from February through August 1997.

All revisions made in the present version are minor, *i.e.*, corrections of typos and the like.

The work was made possible by a Ph.D. grant from the Danish Natural Science Research Council, SNF.

Jakob Rehof
November 14, 1998
Microsoft Research
Redmond, WA, USA

Acknowledgements

First of all, I would like to thank my advisor and friend Fritz (Henglein), who has been a constant source of inspiration and support for many years now. Had he not taken an early interest in my studies, I would not have begun doing research at all, and much of my work, including much work presented in this thesis, was developed jointly with him. I am extremely grateful for his teaching and collaboration. His enormous energy and dedication as well as his combined theoretical and practical sense will always remain a model for me. And, by the way, thanks much for turning me on to type theory – apart from the fact that it somehow connected well with my previous life as a classical scholar, it's been a lot of fun – and sweat!

A very special thanks goes to Alex Aiken whose interest in my work has meant very much to me. A warm thanks also to Manuel Fähndrich and the other guys in the Berkeley program analysis team for sharing with me their work, experience and their company.

I would like to thank Neil Jones and Mads Tofte very much for supporting my work and making it possible for me to visit Stanford University. Also, I would like to state my gratitude towards Bob Paige for having supported my work some years ago.

I am very grateful to John Mitchell who kindly hosted me as a visiting researcher at Stanford. I must thank also my Stanford office mates, Ashish Goel and Uli Stern for having made my office hours so very enjoyable. Thanks to Nikolaj Bjørner and Jens Ulrik Skakkebæk for receiving me and my family so warmly in the States.

For very rewarding discussions on subtyping and other stuff during the past couple of years, I wish to thank also: Anindya Banerjee, Amir Ben-Amram, Marcin Benke, Alexandre Frey, Jesper Jørgensen, Torben Mogensen, Christian Mossin, Francois Pottier, Vaughan Pratt, Alexy Schubert, Morten Heine Sørensen, Valery Trifonov, Pawel Urzyczyn. My sincere apologies to those I may have forgotten to mention in the list.

Thanks to my thesis committee, Dexter Kozen, Michael Schwartzbach and Jerzy Tiuryn, for coming to Copenhagen from both long, short and medium distance, for taking the time to review the thesis, for their feedback and for an enjoyable time in Copenhagen.

Finally, the warmest thanks go to my wife and my children for putting up with all this. While academic research can be very intellectually rewarding to the person who does it, it does not bring the same compensation to those at home.

Contents

Chapter 1

Introduction

The main objective of this thesis is to understand why polymorphic subtype inference systems may be inherently difficult to scale up. This is done through a study of the complexity of *type size*, constraint *simplification* and constraint *entailment* in simple subtyping systems. The remainder of this introduction will explain our motivation in more detail.

1.1 The complexity of subtype inference systems

There are two major aspects of the complexity of subtype inference systems:

1. *The complexity of the typability problem*
2. *The structure and complexity of typings*

The first is the problem to decide, when given a program M as input, whether M has a type in a given subtyping system. An important reason for solving the problem is to obtain information about *type safety* [50] of a program. The second aspect concerns *representing and manipulating typings* efficiently. While the typability problem is fairly well understood by now, the problem of representing and manipulating typings efficiently in subtype inference systems is not. Whereas the combinatorial bottleneck problem in deciding typability is to decide *satisfiability* of subtype constraints in partial orders, one could say that the combinatorial bottleneck problem in representing typings is to decide *entailment* of subtype constraints. The former problem has previously received more attention than the latter.

This thesis mainly investigates the second aspect of the complexity of subtype inference, by studying the two closely related topics:

1. The problem of *subtype simplification*
2. The problem of *subtype entailment*

Simplification aims at presenting typings in an irredundant and succinct form. Deciding entailment is a key problem in simplification and in information extraction from subtyping judgements. The thesis falls in two corresponding parts, Part I studies the former problem, Part II the latter. The emphasis is on algorithmic and complexity aspects. In Part I we explore *the limits of subtype simplification*, in a variety of subtyping systems, regardless of how clever we are at simplifying. In Part II we explore *the cost of subtype simplification* by studying the computational complexity of deciding entailment in a variety of systems. An overall message of the thesis is:

- *Simplification is complex, even in weak subtype logics*, and more generally,
- *Even in simple and well–behaved subtyping systems, typings possess highly complicated structure, which must be controlled in practice*

The work presented in the thesis is intended to be foundational, but it was directly motivated by practical concerns. We seek to understand what features of subtyping systems contribute to their complexity, in the hope that we may be in a better position to identify subtyping systems with good scalability properties in the future.

In the remainder of this introduction we explain why the problems of simplification and entailment are important in subtyping and why they lead to interesting problems of complexity.

1.2 Simple subtyping systems

Simple subtyping [53] arises from the *simple typed λ–calculus* (called λ_s for short) [8, 54] by extending it with a *subsumption* rule of the form

$$[sub] \quad \frac{C, \Gamma \vdash M : \tau \qquad C \rhd \tau \leq \tau'}{C, \Gamma \vdash M : \tau'}$$

meaning that, whenever a program has type τ, then it also has any type τ' larger than τ, with respect to the subtype order $\leq$. A *typing judgement*

$$C, \Gamma \vdash M : \tau$$

expresses that the program M has the type τ under the assumptions contained in C and Γ. The subtyping system is a proof system for deriving typing judgements. A program is typed under a set of *subtyping constraints* C consisting of inequalities of the form $\tau \leq \tau'$, where τ and τ' are type expressions; the constraints express hypotheses about subtype relations which must hold between the types for the program to be well-typed. The set Γ contains type assumptions of the form $x : \tau$, one for each of the free program variables x occurring in M. The relation $C \rhd \tau \leq \tau'$ holds if $\tau \leq \tau'$ somehow follows from the hypotheses in C. There are several possible relations we can use to give meaning to $\rhd$. Whenever $\tau \leq \tau'$ holds, we say that τ is a *subtype* of τ' or that τ' is a *supertype* of τ. As a consequence of subsumption, a subtyping system allows an object of type τ to be type correctly used in any context expecting a supertype of τ.

1.3 Uniqueness and size of principal typings

A very important new problem arising in subtyping systems as opposed to, say, simple typed λ–calculus and ML, is the breakdown of strong uniqueness properties of principal typings. It is well known that principal types in λ_s and ML are unique modulo highly trivial equivalences. In λ_s there exist principal typing judgements of the form $\Gamma \vdash M : \tau$ which are unique up to renaming of type variables [35], and ML has principal quantified type schemes [19, 18] of the form $\forall\vec{\alpha}.\tau$ which are unique up to renaming of bound variables, reordering of quantifiers and dropping of dummy quantifiers. A similarly simple situation does not hold for subtyping systems, where equivalent, principal typings for the same term may differ in highly non–trivial ways. This is basically caused by *constrained types* in subtyping judgements. Intuitively, a principal subtyping judgement of the form

$$C, \Gamma \vdash M : \tau$$

typically prescribes a type τ for M with a *higher degree of freedom* than the corresponding simple principal type. However, the subtyping assumptions appearing in C may be necessary to constrain the type τ to avoid too much freedom. This gives rise to a complicated interplay between type information and constraints. The complications appear to be inherent, and constrained types have always been found to be necessary for the existence of principal typings in subtyping systems [28, 29, 53, 37].

Let us illustrate the points made above with simple examples. In the subtyping system studied by Fuh and Mishra [29], a principal typing of the function $twice = \lambda f.\lambda x.f(f\ x)$ is

$$\{\alpha \leq \beta\}, \emptyset \vdash twice : (\beta \to \alpha) \to (\beta \to \alpha)$$

whereas the principal simple type for *twice* is $(\alpha \to \alpha) \to (\alpha \to \alpha)$. To indicate the nature of the difference, suppose we have type constants `int` and `real` ordered by $\texttt{int} \leq \texttt{real}$ in the subtyping system. Then the type $(\texttt{real} \to \texttt{int}) \to (\texttt{real} \to \texttt{int})$ is one possible type that *twice* has; however, this type is not a substitution instance of the principal simple type. The solution adopted in subtyping systems is to enrich typing judgements with constraint sets that allow types with a higher *degree of freedom* than corresponding simple types[1]. This way, the expected instances can be generated from the type by substitutions that respect the subtyping hypotheses expressed in the constraints. Accordingly, we can generate the instance type $(\texttt{real} \to \texttt{int}) \to (\texttt{real} \to \texttt{int})$ from the principal subtyping shown, because the substitution $\{\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{real}\}$ *satisfies* the constraint set $\{\alpha \leq \beta\}$ in the subtype order $\texttt{int} \leq \texttt{real}$. The constraint set is intuitively necessary, because leaving the type $(\beta \to \alpha) \to (\beta \to \alpha)$ unconstrained would not exclude invalid instances such as $(\texttt{int} \to \texttt{real}) \to (\texttt{int} \to \texttt{real})$.

To see how uniqueness of principal typings breaks down, consider the identity function $\lambda x.x$. In the subtyping system studied by Fuh and Mishra [28], this term has at least the following principal typings:

1. $\{\alpha \to \beta \leq \gamma, \alpha \leq \beta\}, \emptyset \vdash \lambda x.x : \gamma$
2. $\{\alpha \leq \beta\}, \emptyset \vdash \lambda x.x : \alpha \to \beta$
3. $\emptyset, \emptyset \vdash \lambda x.x : \alpha \to \alpha$

It already requires non–trivial reasoning to realize that these typings can all be regarded as equivalent, principal typings for the identity. The reader is invited to sit back for a while and imagine how complicated the situation could become for real programs.

The principal type of a simple typed or an ML–typed program is guaranteed to be the syntactically *shortest* possible type for that program, because any other type can be produced from the principal one by substitution, and

[1] The type $(\beta \to \alpha) \to (\beta \to \alpha)$ has a higher degree of freedom than $(\alpha \to \alpha) \to (\alpha \to \alpha)$, because the former type does not impose the constraint that all its type variables must be the same.

substitution can only increase the size of a type. This is another very significant property which breaks down in subtyping systems, where generality (principality) is no longer synonymous with succinctness. In fact, we will see in this thesis that in some subtyping systems there are terms for which all principal subtyping judgements must be *exponentially* larger than some other typings for the same terms. As a less spectacular but immediately accessible example, consider that the term *twice* above has the instance

$$\emptyset, \emptyset \vdash \mathit{twice} : (\mathtt{real} \to \mathtt{int}) \to (\mathtt{real} \to \mathtt{int})$$

which is shorter than the principal typing shown (we have removed the constraint.)

1.4 Simplification

The situation described in Section 1.3 leads to the problem of *subtype simplification*. Simplifications are transformations on subtyping judgements which aim at removing redundant information from typings. Redundant information can be understood intuitively as *unnecessary degrees of freedom*. In our example typings of the identity $\lambda x.x$, we should clearly prefer the principal typing shown last, because all freedom has been eliminated; the resulting typing is evidently more informative and more succinct than the others.

Even though several type inference algorithms have appeared for several subtyping systems (*e.g.*, [53, 28, 29]), it is generally recognized that the problems mentioned above represent a serious obstacle to practicable, larger scale subtype inference. To quote from [37], "the main problems seem to be that the algorithm is inefficient, and the output, even for relatively simple input expressions, appears excessively long and cumbersome to read". The problem has generated a significant amount of work which aims at *simplifying* constraints in the typings generated by subtype inference algorithms; works addressing the subtype simplification problem include [28, 17, 41, 67, 21, 59, 73, 22, 4, 25]. As is argued in [4], simplification is beneficial for at least three reasons: first, it may speed up type inference, second, it makes types more readable, and, third, it makes the information content of a typing more explicit.

The point about the speed of type inference comes out most forcefully when we consider *polymorphic subtyping*, where ML–style parametric polymorphism [50] is combined with subtyping. In addition to the rule [*sub*], such

systems contain rules for generalization, instantiation and polymorphic `let`:

$$[gen] \quad \frac{C \cup C', \Gamma \vdash M : \tau \ (\alpha_i \notin \mathsf{FV}(C, \Gamma))}{C, \Gamma \vdash M : \forall \vec{\alpha}.C'/\tau}$$

$$[inst] \quad \frac{C, \Gamma \vdash M : \forall \vec{\alpha}.C'/\tau \qquad C \rhd C'\{\vec{\alpha} \mapsto \vec{\tau'}\}}{C, \Gamma \vdash M : \tau\{\vec{\alpha} \mapsto \vec{\tau'}\}}$$

$$[let] \quad \frac{C, \Gamma \vdash M : \sigma \qquad C, \Gamma \cup \{x : \sigma\} \vdash N : \tau}{C, \Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : \tau}$$

Here, type schemes σ of the form $\forall \vec{\alpha}.C/\tau$ are polymorphic *qualified* type schemes [40], allowing instantiation of the quantified variables, provided that the instantiation respects the subtype constraints in C. If the ML type inference algorithm $\mathcal{W}$ is adapted to polymorphic subtyping, then it will typically have a case of the form:

$$\begin{aligned} \mathcal{W}(\Gamma, x) \quad = \quad & \text{if } \Gamma(x) = \forall \vec{\alpha}.C/\tau \\ & \text{then } \langle C\{\vec{\alpha} \mapsto \vec{\beta}\}, \tau\{\vec{\alpha} \mapsto \vec{\beta}\}\rangle \\ & \text{where } \vec{\beta} \text{ are fresh} \end{aligned}$$

The noticeable thing here is that polymorphic instantiation of qualified type schemes requires that fresh versions of the constraint set C get duplicated into each context of polymorphic use of the `let`–bound variables x. This is potentially very costly, and it is generally agreed that simplification prior to instantiation is essential for scalable performance of polymorphic subtype inference [59, 73, 4, 25].

Simplifications must satisfy *soundness* conditions which guarantee that they *preserve the information content of typings*. In this thesis, we take the standard approach of regarding the information content of a typing judgement to be the set of all its *instances*. Let us write $\mathbf{t} \prec_{inst} \mathbf{t'}$ to signify that the subtyping judgement $\mathbf{t'}$ is an instance of the judgement $\mathbf{t}$. Accordingly, two subtyping judgements $\mathbf{t}$ and $\mathbf{t'}$ for the same term are considered equivalent, written $\mathbf{t} \approx_{inst} \mathbf{t'}$, if and only if they have the same instances, or, equivalently, if and only if they are instances of each other. The soundness condition for a transformation $\mapsto$ on judgements then becomes the condition

$$\mathbf{t} \mapsto \mathbf{t'} \Rightarrow \mathbf{t} \approx_{inst} \mathbf{t'}$$

The condition guarantees that the transformation $\mapsto$ loses no typing power; in particular, a sound transformation will preserve principality of typings.

1.5 Instance

It turns out that there are several meaningful definitions of $\prec_{inst}$ in subtyping systems. A natural requirement on any instance relation is formulated by Hoang and Mitchell in [37]. Here, an instance relation $\prec_{inst}$ is called *sound* if and only if we have

1. The relation $\prec_{inst}$ is independent of terms, *i.e.*, $C, \Gamma \vdash M : \tau \prec_{inst} C', \Gamma' \vdash M : \tau'$ if and only if $C, \Gamma \vdash N : \tau \prec_{inst} C', \Gamma' \vdash N : \tau'$ for all terms N.

2. The relation $\prec_{inst}$ preserves derivability of judgements, *i.e.*, if $C, \Gamma \vdash M : \tau$ is a derivable well–typing of M and $C, \Gamma \vdash M : \tau \prec_{inst} C', \Gamma' \vdash M : \tau'$, then $C', \Gamma' \vdash M : \tau'$ is again a derivable well–typing of M.

While the second requirement hardly needs explanation, the first one is perhaps more subtle. Even so, it is natural and it is satisfied by all known instance relations in any known type system. It effectively says that a notion of instance should only depend upon the *logical* rules of the type system. Here, a rule is called logical, if it can be applied to a term regardless of the structure of the term. In simple subtyping, the only logical rule is [*sub*].

As a consequence of the soundness properties above, a notion of instance for simple subtyping will depend in an essential way on the relation $\rhd$ used in the subsumption rule. There are several possible choices for $\rhd$. The relation most widely used in earlier formulations of subtyping [53] is a syntactic proof relation $\vdash_P$. Subsumption judgements $C \vdash_P \tau \leq \tau'$ mean that $\tau \leq \tau'$ is a provable consequence of the hypotheses C. The proof system assumes a finite poset P of base types (type constants), such as, *e.g.*, $\texttt{int} \leq \texttt{real}$. Typically, the relation $\vdash_P$ has been very weak, axiomatizing only the general partial order properties (reflexivity, transitivity and anti–symmetry of $\leq$) together with rules that allow the order on P to be lifted from base types to constructed types, including the *contra–variant* rule [53] for ordering of function types:

$$[arrow] \quad \frac{C \vdash_P \tau_1' \leq \tau_1 \qquad C \vdash_P \tau_2 \leq \tau_2'}{C \vdash_P \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

The first instance relation suggested for simple subtyping was given by Mitchell [53] and used the relation $\vdash_P$ in the subsumption rule. The relation, called $\prec_{weak}$ here, is defined by setting $C, \Gamma \vdash M : \tau \prec_{weak} C', \Gamma' \vdash M : \tau'$ if and only if there exists a substitution S such that

1. $C' \vdash_P S(C)$
2. $\tau' = S(\tau)$
3. $S(\Gamma) \subseteq \Gamma'$

An important property of $\prec_{weak}$ is that the type system has the *principality property* with respect to it: any typable term has a principal typing with respect to $\prec_{weak}$, see [53, 29]. The relation $\prec_{weak}$ is not very interesting, though, as a foundation for simplification, since it does not validate even quite simple typing transformations. For instance, one cannot eliminate any variable from the typing $\{\alpha \leq \beta\}, \emptyset \vdash_P \lambda x.x : \alpha \to \beta$ without losing principality with respect to $\prec_{weak}$, due to the requirement $\tau' = S(\tau)$.

This inspired Fuh and Mishra [28] to introduce a more powerful notion of instance, called "lazy instance", which was still based on the proof relation $\vdash_P$, but which exploited the subtype logic to a larger degree. The aim was to obtain a larger instance relation (relating more typings) in order to support more powerful simplifications. The rationale is that, if the relation $\prec_{inst}$ is large, then the condition $\mathbf{t} \mapsto \mathbf{t}' \Rightarrow \mathbf{t} \approx_{inst} \mathbf{t}'$ allows more powerful simplifications ($\mapsto$). The lazy instance relation will be denoted $\prec_{syn}$ here[2], and was defined by setting $C, \Gamma \vdash M : \tau \prec_{syn} C', \Gamma' \vdash M : \tau'$ if and only if there exists a substitution S such that

1. $C' \vdash_P S(C)$
2. $C' \vdash_P S(\tau) \leq \tau'$
3. $\mathcal{D}(\Gamma) \subseteq \mathcal{D}(\Gamma')$ and $\forall x \in \mathcal{D}(\Gamma).\ C' \vdash_P \Gamma'(x) \leq S(\Gamma(x))$

where $\mathcal{D}(\Gamma)$ is the domain of the assumption set Γ, regarded as a map from term variables to types. Clearly, $\prec_{weak} \subseteq \prec_{syn}$.

Intuitively, $\prec_{syn}$ allows some subtyping constraints on the type of a program M to be deferred to the (unknown) future contexts of use of M. This validates the adoption of the judgement $\emptyset, \emptyset \vdash \lambda x.x : \alpha \to \alpha$ as principal for the identity. Often, a simplification step can be expressed as the application to a judgement of a substitution which eliminates variables by collapsing them. For example, using $\prec_{syn}$, we can validate the transformation

$$\{\alpha \leq \beta\}, \emptyset \vdash \lambda x.x : \alpha \to \beta \quad \mapsto \quad \emptyset, \emptyset \vdash \lambda x.x : \alpha \to \alpha$$

[2]We call $\prec_{syn}$ a *syntactic* relation, because it is still based on the proof system for $\vdash_P$

by applying the substitution $\{\beta \mapsto \alpha\}$ and realizing that the resulting judgement is equivalent to the original one with respect to $\approx_{syn}$, at each simplification step:

$$\begin{array}{ll} \{\alpha \leq \beta\}, \emptyset \vdash \lambda x.x : \alpha \to \beta & \mapsto \\ \{\alpha \leq \alpha\}, \emptyset \vdash \lambda x.x : \alpha \to \alpha & \mapsto \\ \emptyset, \emptyset \vdash \lambda x.x : \alpha \to \alpha & \end{array}$$

Here, the second judgement arises from the first by mapping α and β to α, thereby transforming the constraint set to the trivial constraint $\alpha \leq \alpha$, which can be eliminated entirely.

1.6 Entailment

It is a natural next step, in the process of empowering simplifications, to use more powerful relations than $\vdash_P$ to play the rôle of $\rhd$ in the type system and in the notions of instance. Since the weakness of $\vdash_P$ lies in the fact that it does not exploit the algebraic properties of the specific order structure generated by P, it is natural to turn to a model theoretic notion of *entailment*, leading to what is sometimes called "model theoretic" or "semantic" subtyping frameworks. This development took place in recent years both within the field of subtyping and set constraint based program analysis.

If C is a constraint set and ϕ is an inequality (such as a subtype inequality or a set inclusion), we say that C *entails* ϕ, written $C \models \phi$, if every assignment of meanings to expressions that satisfies all the constraints in C also satisfies ϕ; an inequality is satisfied under the assignment, if it is true in the intended model. The intended model for subtyping constraints consists of ordered ground types (and for set constraints, the powerset lattice over a Herbrand universe.)

A main motivation in entailment based subtyping is to support, justify and reason about powerful simplification. As a very simple example, suppose that C is the set of subtyping constraints $\{\alpha \leq \texttt{int}, \beta \leq \texttt{bool}, \gamma \leq \alpha, \gamma \leq \beta\}$; we assume that `int` and `bool` are incomparable types, and that there is a smallest type, denoted $\bot$, and no other base types. Then we have $C \models \gamma = \bot$. Based on this entailment, we might simplify C by substituting $\bot$ for γ, yielding $\{\gamma = \bot, \alpha \leq \texttt{int}, \beta \leq \texttt{bool}, \bot \leq \alpha, \bot \leq \beta\}$; since $\bot \leq \delta$ is true no matter what δ is, we can simplify further, yielding just $C' = \{\gamma = \bot, \alpha \leq \texttt{int}, \beta \leq \texttt{bool}\}$. The justification of simplifying C to C' is that $C \models C'$ and $C' \models C$, *i.e.*, C and C' are logically equivalent. Using instance relations such as $\prec_{syn}$ based on $\models$ rather than $\vdash_P$, we can typically reduce

the constraint set further, leaving just the set $\{\alpha \leq \texttt{int}, \beta \leq \texttt{bool}\}$ after suitable substitutions on the entire typing judgement.

Recent work in subtyping systems and in set constraints using notions of constraint entailment includes [3, 14, 59, 73, 22, 4, 25, 48, 13].

With this step to entailment based subtyping taken, we have arrived at the simplification frameworks under study in this thesis. Our reference type system will be one in which the subsumption rule [*sub*] uses various forms of entailment relations ($\models$) in the rôle of $\rhd$.

1.7 Main problems and results

We are now in a position to explain what the main problems studied in this thesis are, and what we have achieved.

The structure and complexity of principal typings (Part I)

Since the objective of simplification is to eliminate unnecessary degrees of freedom in typings, a natural way to test the power of a simplification framework is to ask:

- *How many distinct variables must be present in a principal typing, in the worst case, regardless of how clever we are at simplifying?*

Here, the number of distinct variables is taken as the measure of freedom, and simplification is viewed as a variable elimination procedure (see also [4] for this view.)

In Part I of the thesis, we will answer the question above for a variety of simplification frameworks, based on notions of instance of increasing strength. We show that all known instance relations lead to worst case *exponential* asymptotic growth of the number of distinct variables that must be present in principal typings in *atomic* subtyping systems. This holds no matter how clever we are at simplifying typings relative to a given instance relation. The number of variables is measured as a function of the size of the terms being typed. Atomic systems allow only type variables and constants to occur in constraint sets.

We argue that the exponential growth in the atomic case also characterizes the degree of freedom of typings inherent in non–atomic systems, because such systems only bypass exponential growth by keeping information implicit which is forced to be explicit in atomic systems.

We compare a series of instance relations with respect to the simplifications they validate, and we find that they can be separated by exponential gaps, in the sense that simplifications under stronger relations can, in the extreme, yield exponential compression of typings in comparison to weaker relations, even though none of the instance relations guarantee sub–exponential size of principal typings.

The question above is closely related to the question of *type size*. It is well–known that principal types in λ_s grow exponentially, in the worst case, under textual representation but at most linearly under *dag*–representation (see [42]), and principal ML types grow doubly exponentially under textual representation but at most exponentially under *dag*–representation (see [42] and [54, Chapter 11.3]). Here, type size is measured as a function of the size of the term being typed. The common characteristic of λ_s and ML in this respect is that in both systems dag–representation yields exponential compression of types. The question of asymptotic worst case type size is not so well understood for subtyping systems. The only previous result in this area was given by Hoang and Mitchell, who showed that the size of constraint sets must grow at least linearly in the size of terms for principal typings with respect to any *sound* notion of instance (as defined in Section 1.5).

Our results show that, relative to any known instance relation, subtyping systems are characterized by the absence of compression under dag–representation. This is a reflection of a fundamental difference between subtyping systems and systems such as λ_s and ML, which are based on equality constraints (in the sense of [74]) rather than inequality constraints. Another manifestation of this difference is that, in subtyping systems, it is typically much more difficult to present principal typings succinctly than it is to decide the typability problem. In the other systems, these problems have the same complexity.

The study of type–size for subtyping systems is considerably more complicated than for systems such as λ_s and ML, because of the lack of strong uniqueness properties of principal typings (Section 1.3), and our results require new methods.

Part I of the thesis extends results published by the author in [65].

The complexity of entailment (Part II)

Verifying that a simplification step $\mapsto$ is sound amounts to checking that the condition

$$\mathbf{t} \mapsto \mathbf{t}' \Rightarrow \mathbf{t} \approx_{inst} \mathbf{t}'$$

is satisfied. With instance relations $\prec_{inst}$ based on entailment, this check essentially involves deciding the *entailment problem*:

- *Given a constraint set C and types τ and τ', is it the case that $C \models \tau \leq \tau'$?*

If simplification is automated, then the simplification algorithm typically has to decide entailment problems in order to justify (or reject) a potential simplification step. It is therefore interesting to develop entailment algorithms and to know the computational complexity of deciding entailment.

In Part II of the thesis, we study the computational complexity of deciding entailment over a variety of subtype orders, viewed as ordered structures of labeled trees. We confine ourselves to considering simple and very well–behaved structures, generated from *lattices* of base types, for all of which the *satisfiability* problem is in PTIME.

We prove that *non–structural* subtype entailment is PSPACE-hard, both for finite trees (simple types) and infinite trees (recursive types). For the *structural* subtype ordering we prove that subtype entailment over finite trees is coNP–complete and entailment over infinite trees is PSPACE-complete. Our proof methods centre on viewing constraint sets as nondeterministic finite automata and may have independent interest.

These are the first complexity–theoretic separation results that show that, informally, non–structural subtype entailment is harder than structural entailment, and recursive entailment is harder than non–recursive entailment. More precisely, assuming NP $\neq$ PSPACE, we show that both finite non–structural and recursive structural subtype entailment are strictly harder than finite structural subtype entailment.

The results show that passing from the satisfiability problem to the entailment problem exposes an entirely different structure of complexity, since the satisfiability problem is in PTIME for all structures considered. Since the satisfiability problem is the bottleneck in deciding typability, our results again confirm that presenting simplified typings is much harder than deciding typability in subtyping systems.

Summaries of core results presented in Part II of the thesis have been published jointly with Fritz Henglein, in [33, 34].

A personal note on simplicity

It is a common characteristic of all systems considered in this thesis that they are *simple* and logically *weak*. In Part I, this is manifested especially

in the decision to consider simple subtyping systems, and in the fact that our constraint languages have *no algebraic operators* at all, only inequalities between terms built from variables, constants and syntactic constructors will be allowed at any time. In Part II, simplicity is manifested in the decision to consider only *lattice–based* orders. This is the result of a very conscious decision. In general, most of my work is driven by a desire to discover program logics that have a chance to scale up to very large programs. To me, at least, a central message of this thesis is that, when seen through the spectrum of the problems of simplification and entailment,

- *Even the simplest and weakest subtype logics are very complex*

and the lesson I personally draw from this is that the logics considered in this thesis may have to be further limited, in some cases to a very considerable degree, before they have a chance to become useful in the setting of very large scale program analysis.

Ultimately, my interest lies in polymorphic subtype based program analysis. This is where the problems studied in this thesis are most forcefully present and have the greatest practical relevance. The decision to consider only simple subtyping was made, because I believe that the most fundamental problems in subtype simplification are already present and are seen in sharper focus at this level. Moreover, highly expressive formalisms have already been studied by others. Notably, a very impressive body of knowledge has been obtained for set constraint languages containing full boolean machinery. Given this, I decided that it might be useful to contribute to our understanding of very weak subtype logics.

Chapter 2

Subtype orders and subtyping systems

In this chapter we introduce basic concepts from the literature about subtype orders (Section 2.1), we define the subtyping systems under study (Section 2.2), and we recall some previous results from the theory of subtyping (Section 2.3), which will be used pervasively in the thesis. Section 2.3 will also point to a few background results provided in this thesis. The reader may wish to just skim quickly through the chapter in order to return to it for more detailed reference later when needed.

2.1 Subtype order and subtype constraints

The distinctive feature of subtyping systems is that they exploit order structures on types. At the level of generality needed for this thesis, types are best viewed as labeled trees. We will therefore begin by defining subtype orders in the form of ordered tree structures.

2.1.1 Ordered trees

Let Σ be the ranked alphabet of *constructors*, $\Sigma = B \cup \{\times, \to\}$; here $B = (P, \leq_P)$ is a finite poset of *constants* (constructors of arity 0, to be thought of as base types, such as, *e.g.*, `int`, `real`) and $\times, \to$ are binary constructors. The order on P defines the subtype ordering on type constants in the standard way (see, *e.g.*, [53] for a full introduction.)

Let $\mathbf{A}$ be the alphabet $\mathbf{A} = \{f, s, d, r\}$, and let $\mathbf{A}^*$ denote the set of

finite strings over $\mathbf{A}$; elements of $\mathbf{A}^*$ are called *addresses*. Typical elements of B are ranged over by b, typical elements of $\mathbf{A}$ are ranged over by a, and typical elements of $\mathbf{A}^*$ are ranged over by w. We consider types as *labeled trees* in the style of [45].

A *tree* over Σ is a partial function

$$t : \mathbf{A}^* \to \Sigma$$

with domain $\mathcal{D}(t)$, satisfying the following conditions:

- $\mathcal{D}(t)$ is non-empty and prefix-closed
- if $t(w) = \times$, then $\{a \in \mathbf{A} \mid wa \in \mathcal{D}(t)\} = \{f, s\}$
- if $t(w) = \to$, then $\{a \in \mathbf{A} \mid wa \in \mathcal{D}(t)\} = \{d, r\}$
- if $t(w) \in B$, then $\{a \in \mathbf{A} \mid wa \in \mathcal{D}(t)\} = \emptyset$

A tree t is *finite* (resp. *infinite*) if and only if $\mathcal{D}(t)$ is a finite (resp. infinite) set. If $w \in \mathcal{D}(t)$ and w is not a proper prefix of any string in $\mathcal{D}(t)$, then w is called a *leaf address* of t (*i.e.*, $t(w) \in P$); we write $\mathsf{Lf}(t)$ to denote the set of leaf addresses in t. If $w \in \mathcal{D}(t)$ and w is not a leaf address, then w is said to be an *interior address* of t; we write $\mathsf{In}(t)$ to denote the set of interior addresses in t. Notice that, whenever w is an interior address in t, then $t(w)$ is completely determined by $\mathcal{D}(t)$, *e.g.*, $t(w) = \times$ if and only if $\{wf, ws\} \subseteq \mathcal{D}(t)$.

We let $\mathcal{T}_\Sigma$ denote the set of trees over Σ. For a partial order $\leq$, we let $\leq^0$ denote the order $\leq$ itself and $\leq^1$ the reversed relation, $\geq$. For $w \in \mathbf{A}^*$, we define the *polarity* of w, denoted $\pi(w)$, to be 0 if w contains an even number of d's and $\pi w = 1$ otherwise. We write $\leq^w$ as a shorthand for $\leq^{\pi(w)}$. The reversal of order in accordance with polarity captures contravariance of the subtype order with respect to the function space constructor $\to$ (see [45].) If Σ is equipped with a partial order $\leq_\Sigma$, we induce a partial order on $\mathcal{T}_\Sigma$ by setting

$$t \leq t' \text{ if and only if } \forall w \in \mathcal{D}(t) \cap \mathcal{D}(t').\ t(w) \leq_\Sigma^w t'(w)$$

Let $\mathcal{T}_\Sigma^{\mathrm{F}}$ be the set of finite trees over Σ. Then $\mathcal{T}_\Sigma^{\mathrm{F}}$ inherits the order on $\mathcal{T}_\Sigma$, so that the ordered structure $\mathcal{T}_\Sigma$ is a conservative extension of $\mathcal{T}_\Sigma^{\mathrm{F}}$.

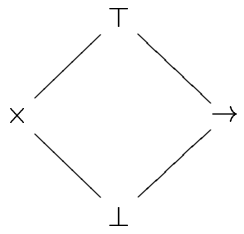


Figure 2.1: Lattice Σ for non–structural subtype order

2.1.2 Type expressions

Let $\mathcal{V}$ be a denumerable set of variables, distinct from elements of Σ. *Type expressions* (also called *terms* or just *types*) over Σ are *finite* trees in $\mathcal{T}_{\Sigma\cup\mathcal{V}}$, where elements of $\mathcal{V}$ are regarded as constructors of arity 0 (however, only members of Σ of arity 0 are called constants); type expressions are ranged over by τ. Type expressions that are either constants or variables are referred to as *atoms* or *atomic types* (they have no complex syntactic structure.) Atoms are ranged over by A. The set of terms is denoted $\mathcal{T}_\Sigma(\mathcal{V})$. Terms can be defined by the grammar

$$\begin{array}{lcl} \tau & ::= & A \mid \tau \times \tau' \mid \tau \to \tau' \\ A & ::= & \alpha \mid b \end{array}$$

where A ranges over atoms, α ranges over $\mathcal{V}$ and b ranges over $B = (P, \leq_P)$.

2.1.3 Non–structural subtype order

The *non–structural* subtype order is obtained by fixing P to be the two–point lattice, $P = \{\bot, \top\}$, and Σ to be the set $\{\times, \to, \top, \bot\}$ organized as a lattice by the order $\bot \leq \sigma$, $\sigma \leq \top$ for $\sigma \in \Sigma$. The Hasse–diagram of Σ is shown in Figure 2.1. See [45] for further details. This organizes $\mathcal{T}_\Sigma$ as a complete lattice under the non–structural order. We will sometimes use a special name for the structure $\mathcal{T}_\Sigma$ under this ordering. In such cases, we will denote the set of non–structurally ordered trees by $\mathcal{T}_\Sigma[n]$, and the set of finite trees under the non–structural order by $\mathcal{T}_\Sigma^{\mathrm{F}}[n]$.

A characteristic feature of the non–structural order is the ability to compare trees with different "shapes" (domains). For instance, one has $\bot \leq t$ and $t \leq \top$ for all t.

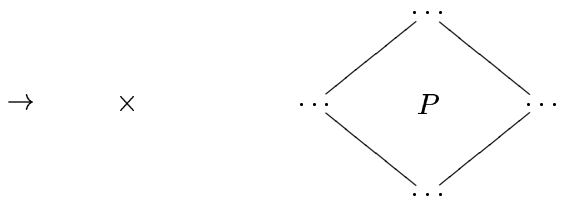


Figure 2.2: Ordered set Σ for structural subtype order

2.1.4 Structural subtype order

The *structural* subtype order is obtained by fixing Σ to be $\Sigma = P \cup \{\times, \rightarrow\}$, where the order $\leq_P$ on P is extended to Σ; that is, by adding $\times$ and $\rightarrow$ as new, incomparable elements to P. The Hasse–diagram of Σ is shown in Figure 2.2. We will sometimes use a special name for the structure $\mathcal{T}_\Sigma$ under this ordering. In such cases, we will denote the set of structurally ordered trees by $\mathcal{T}_\Sigma[s]$, and the set of finite trees under the structural order by $\mathcal{T}_\Sigma^{\mathrm{F}}[s]$.

A characteristic feature of the structural order is that, for any two trees t and t' to be comparable, t and t' must have the same "shape" (domain); the ordering of such trees t and t' is reducible to the way corresponding constants at the leaves of t and t' are ordered in P.

Lemma 2.1.1 *In the structural subtype order, we have $t \leq t'$ if and only if the following conditions hold:*

1. $\mathcal{D}(t) = \mathcal{D}(t')$
2. *for every interior address w of t and t', one has $t(w) = t'(w)$*
3. *for every leaf address w of t and t', one has $t(w) \leq_P^w t'(w)$*

PROOF The "if" direction is obvious from the definitions. For the "only if" direction, assume $t \leq t'$ and that (w.l.o.g.) there is a string w of shortest length such that $w \in \mathcal{D}(t) \setminus \mathcal{D}(t')$. Write $w = w'a$, $a \in \mathbf{A}$ (this is possible, since the empty string is in both domains), then $w' \in \mathcal{D}(t) \cap \mathcal{D}(t')$ with $t(w') \leq_\Sigma t'(w')$; the requirements on the order $\leq_\Sigma$ together with the requirements on tree domains then show that $t(w')$ and $t'(w')$ must be the same binary constructor, and hence $w'a \in \mathcal{D}(t)$ if and only if $w'a \in \mathcal{D}(t')$,

a contradiction. This shows that $\mathcal{D}(t) = \mathcal{D}(t')$, and the other conditions follow easily from the definition of $\leq_\Sigma$. □

Lemma 2.1.1 shows that we could as well have defined the order $\leq$ on trees by the three conditions of the lemma, and henceforth we shall tacitly assume this.

Suppose that the poset (called P above) of constants happens to be a lattice, L. For each tree t in $\mathcal{T}_\Sigma[s]$, let L_t be the set of trees $\{t' \in \mathcal{T}_\Sigma[s] \mid \mathcal{D}(t) = \mathcal{D}(t')\}$, and let $\mathcal{L} = \{L_t \mid t \in \mathcal{T}_\Sigma[s]\}$. Using Lemma 2.1.1 it is easy to verify that each L_t is a complete lattice (under the order inherited from $\mathcal{T}_\Sigma[s]$). Moreover, the ordered structure $\mathcal{T}_\Sigma[s]$ is the disjoint union of all the distinct L_t:

$$\mathcal{T}_\Sigma[s] = \biguplus_{L \in \mathcal{L}} L$$

2.1.5 Summary of structures

It is convenient to summarize in one place all the ordered structures we shall be working with:

P	=	A finite partial order of type constants
L	=	A finite lattice of type constants
$\mathcal{T}_\Sigma[s]$	=	General trees with structural order
$\mathcal{T}_\Sigma[n]$	=	General trees with non–structural order
$\mathcal{T}_\Sigma^{\mathrm{F}}[s]$	=	Finite trees with structural order
$\mathcal{T}_\Sigma^{\mathrm{F}}[n]$	=	Finite trees with non–structural order

To avoid heavy notation, we shall sometimes avoid using these explicit names. In such cases we will stipulate, within a given context of discussion, that $\mathcal{T}_\Sigma$ denote a particular one of the structures mentioned above.

2.1.6 Subtype inequalities and constraint sets

A *constraint set* is a finite set of formal *inequalities* of the form $\tau \leq \tau'$, where τ and τ' are type expressions. We sometimes let ϕ and ψ range over inequalities, when we are not interested in the form of the component type expressions. We write $\mathsf{Var}(C)$ to denote the set of variables that occur in C. The notation $\leq^w$ may be extended to formal inequalities (denoting reversed formal inequality if $\pi(w) = 1$.) We write $\tau = \tau'$ (formal equality) as a shorthand for the simultaneous inequalities $\tau \leq \tau'$ and $\tau' \leq \tau$.

Type expressions and constraints will be interpreted in the various structures of ordered trees, by mapping type variables to trees. Under such a mapping, type expressions denote trees and constraints are predicates over ordered trees. By slight abuse of words, a constraint set C is called *structural* (resp. *non–structural*) if its intended interpretation is in structurally (resp. non–structurally) ordered trees. Notice that this has nothing to do with the syntactic form of constraint sets (which is in all cases the same), only the intended model.

A constraint set C is called *atomic* if every inequality in C has the form $A \leq A'$, *i.e.*, only atomic type expressions occur in C. In the literature, atomic inequalities are sometimes called *flat* inequalities.

2.1.7 Recursive and finite subtype orders

Both structural and non–structural subtype constraints have finite and infinite variants, according to whether variables range over finite or infinite trees; in the infinite case, we talk about *recursive* subtype constraints. Again, the language of subtyping constraints remain the same, viz. inequalities $\tau \leq \tau'$ between simple type expressions, whether we consider the model of infinite trees or the model of finite trees. There will be constraints such as for instance $\alpha = \alpha \times \beta$ which have solutions in the former case but not in the latter.

In the case of finite trees, we can define the subtype order by simple logical systems, and it may sometimes be convenient to refer to that formulation. For the finite, structural case the subtype order can be defined by the rules given in Figure 2.3. The subtype logic defines provable judgements of the form $C \vdash_P \tau \leq \tau'$, meaning that the inequality $\tau \leq \tau'$ is a provable consequence of the subtype assumptions in constraint set C. For the finite, non–structural case, one adds the axiom schemes $\bot \leq \tau$ and $\tau \leq \top$ to the rules of Figure 2.3.

2.1.8 Valuation, satisfaction, entailment

In different contexts we shall fix one of the ordered structures mentioned in Section 2.1.5 to be the current constraint model. We then consider valuations, satisfaction and entailment relative to this fixed model. Let $\mathcal{M}$ be any one of the ordered tree structures mentioned above.

A *valuation* is a ground substitution, *i.e.*, a map $v : \mathcal{V} \to \mathcal{M}$, extended canonically to a substitution $v : \mathcal{T}_\Sigma(\mathcal{V}) \to \mathcal{T}_\Sigma$.

$$[const] \quad C \vdash_P b \leq b', \text{ provided } b \leq_P b'$$

$$[ref] \quad C \vdash_P \tau \leq \tau$$

$$[hyp] \quad C \cup \{\tau \leq \tau'\} \vdash_P \tau \leq \tau'$$

$$[trans] \quad \frac{C \vdash_P \tau \leq \tau' \quad C \vdash_P \tau' \leq \tau''}{C \vdash_P \tau \leq \tau''}$$

$$[prod] \quad \frac{C \vdash_P \tau_1 \leq \tau_1' \quad C \vdash_P \tau_2 \leq \tau_2'}{C \vdash_P \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}$$

$$[arrow] \quad \frac{C \vdash_P \tau_1' \leq \tau_1 \quad C \vdash_P \tau_2 \leq \tau_2'}{C \vdash_P \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

Figure 2.3: Subtype logic for finite structural subtyping over an arbitrary poset P

A valuation *satisfies* a constraint $\tau \leq \tau'$, written $v \models \tau \leq \tau'$, if $v(\tau) \leq v(\tau')$ is true in the (appropriate, structural or non–structural) ordered structure of (finite or infinite) trees, $\mathcal{T}_\Sigma$. We write $v \models C$ if v satisfies all the constraints in C, *i.e.*,

$$\forall \tau \leq \tau' \in C.\ v \models \tau \leq \tau'$$

A constraint set C *entails* a constraint $\tau \leq \tau'$, written $C \models \tau \leq \tau'$, if every valuation satisfying all the constraints in C also satisfies the constraint $\tau \leq \tau'$, in symbols

$$\forall v : \mathcal{V} \to \mathcal{M}.\ v \models C \Rightarrow v \models \tau \leq \tau'$$

Often we shall leave $\mathcal{M}$ implicit in our notation, as is done above, but sometimes we will make it explicit, by writing $\models_{\mathcal{M}}$.

There are standard logical decision problems associated with the notions of satisfaction and entailment. The *satisfiability problem* is

- Given a constraint set C as input, determine whether there exists a valuation satisfying C (in the appropriate structure of ordered trees)

The *entailment problem* is

- Given a constraint set C and variables α, β as input, determine whether $C \models \alpha \leq \beta$ holds (in the appropriate structure of ordered trees)

Notice that the problem of deciding $C \models \tau \leq \tau'$ can be stated in the above mentioned form, modulo a trivial transformation, namely as the problem $C \cup \{\alpha = \tau, \beta = \tau'\} \models \alpha \leq \beta$, where α and β are fresh variables not occurring in C (recall that we use formal equality, $\tau = \tau'$, as an abbreviation for the two inequalities $\tau \leq \tau'$ and $\tau' \leq \tau$).

We say that a set C entails another set C' if and only if $C \models \tau \leq \tau'$ for all $\tau \leq \tau'$ in C'. We say that C and C' are *logically equivalent*, written $C \sim C'$, if and only if $C \models C'$ and $C' \models C$.

2.2 Subtyping systems

The subtyping systems considered in this thesis all arise from extending the simply typed λ–calculus [8] (λ_s for short) by a rule of *subsumption*, as explained in Section 1.2. Figure 2.4 gives the rules of our standard system. This system will be referred to as $\lambda_{\leq}$ for short. The subsumption rule, called [*sub*] in Figure 2.4, uses the entailment relation $\models$ to define the valid subsumptions. The rules in Figure 2.4 is a logic for deriving judgements of the form

$$C, \Gamma \vdash_P M : \tau$$

Here, C is a constraint set containing formal inequalities of the form $\tau \leq \tau'$ between simple type expressions; Γ is a set of type assumptions of the form $x : \tau$, one assumption for every variable x free in M. The rule [*base*] assigns to a term–constant c a pre–defined type of c, called TypeOf(c). The system is often parametric in a finite poset P of base types, as signified by the subscripted turnstile, $\vdash_P$. We say that a judgement is *derivable* if it has a proof in the system. Using the rules in Figure 2.4, we can define a number of different variants of subtyping by fixing any one of the structures summarized in Section 2.1.5 to be the intended constraint model. The intention is that a typing judgement $C, \Gamma \vdash_P M : \tau$ represents a well–typing of the λ–term M if and only if

1. the judgement is derivable, and
2. the constraint set C of the judgement is satisfiable in the intended structure

$$[var] \qquad C, \Gamma \cup \{x : \tau\} \vdash_P x : \tau$$

$$[base] \qquad C, \Gamma \vdash_P c : \mathrm{TypeOf}(c)$$

$$[abs] \qquad \frac{C, \Gamma \cup \{x : \tau\} \vdash_P M : \tau'}{C, \Gamma \vdash_P \lambda x.M : \tau \to \tau'}$$

$$[app] \qquad \frac{C, \Gamma \vdash_P M : \tau \to \tau' \quad C, \Gamma \vdash_P N : \tau}{C, \Gamma \vdash_P M\ N : \tau'}$$

$$[sub] \qquad \frac{C, \Gamma \vdash_P M : \tau \quad C \models \tau \leq \tau'}{C, \Gamma \vdash_P M : \tau'}$$

Figure 2.4: Subtyping system

We then have the following catalogue of different subtyping systems, all of which have been studied in the literature on subtyping (main references are given below):

- *Atomic subtyping* is obtained by requiring that only atomic constraint sets may occur in typings and fixing the constraint model to be a finite poset P of base types. The structure of P is significant for the properties of the system. In particular, if P is a lattice, then we talk about *atomic subtyping over a lattice.*

- *Structural subtyping* is obtained by fixing the constraint model to be $\mathcal{T}_\Sigma[s]$ or $\mathcal{T}_\Sigma^{\mathrm{F}}[s]$. In the former case, we have *structural recursive subtyping*, and in the latter case we have *finite structural subtyping.* In each case, the structure $\Sigma = B \cup \{\to, \times\}$ is significant. In particular, if $B = (L, \leq_L)$ is a *lattice* of base types, we have *structural subtyping over a lattice* (in either the recursive or the finite variant).

- *Non–structural subtyping* is obtained by fixing the constraint model to be $\mathcal{T}_\Sigma[n]$ or $\mathcal{T}_\Sigma^{\mathrm{F}}[n]$. In the former case, we have *non–structural recursive subtyping*, and in the latter case we have *non–structural finite subtyping.*

Many of the systems discussed in the literature have been based on a syntactic proof system for derivable judgements, which arises from the rules in

Figure 2.4 by exchanging the relation $\models$ in the subsumption rule [*sub*] with the relation $\vdash_P$ defined in Figure 2.3. In the syntactic version, atomic and finite structural subtyping was introduced and studied by Mitchell [51, 53] and subsequently many others, including Fuh and Mishra [28, 29]. Structural recursive subtyping arises as a natural generalization and has been studied by, among others, Tiuryn and Wand [72]. Non–structural recursive subtyping was introduced by Amadio and Cardelli [5, 6], and its finite version corresponds to Thatte's partial types [69, 70]; type inference with non–structural subtyping has been investigated in a series of papers authored (in various combinations) by Kozen, O'Keefe, Palsberg, Schwartzbach and Wand [45, 44, 57, 56, 58].

2.3 Basic concepts and properties

This section concerns background information about basic concepts and properties of subtyping systems, which will be pervasive. The properties are mainly given for the non–recursive, structural case. Some of the properties will be generalized to other systems later.

The following concepts will be pervasive:

Definition 2.3.1 (Weak unifiability) If C is a constraint set, $\star$ is an arbitrary, fixed constant and τ a type expression, let $\tau^\star$ be the type expression which arises from τ by replacing all constants in τ by the same constant $\star$. Define the set E_C by

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

We say that C is *weakly unifiable* if and only if E_C is unifiable. In the case of *finite, structural* subtyping, unification is tacitly understood to be non–circular (*i.e.*, it has the occurs–check), whereas unification is implicitly taken to be circular for the case of *recursive, structural* subtyping. We let U_C denote the most general unifier for E_C (under circular or non–circular unification, as appropriate). □

Definition 2.3.2 (Constraint closure) A constraint set C is said to be *closed* if and only if the following conditions are satisfied:

- (transitivity)
  $\tau_1 \leq \tau_2 \in C, \tau_2 \leq \tau_3 \in C \Rightarrow \tau_1 \leq \tau_3 \in C$

- (product decomposition)
  $\tau_1 \times \tau_2 \leq \tau_3 \times \tau_4 \in C \Rightarrow \{\tau_1 \leq \tau_3, \tau_2 \leq \tau_4\} \subseteq C$
- (arrow decomposition)
  $\tau_1 \to \tau_2 \leq \tau_3 \to \tau_4 \in C \Rightarrow \{\tau_3 \leq \tau_1, \tau_2 \leq \tau_4\} \subseteq C$

We define the *closure of* C, denoted $\mathsf{Cl}(C)$, to be the least closed constraint set containing C. We let C^* denote the transitive closure of C (the least set containing C closed under the transitivity rule above.) □

Definition 2.3.3 (Consistency) There are standard notions of consistency for subtyping constraint sets (see, *e.g.*, [71, 59]), but the definition for non–structural sets is slightly different from the one used for structural sets. A *non–structural* constraint set C is called *consistent* if and only if at least one of the following conditions are satisfied for every inequality $\tau \leq \tau' \in \mathsf{Cl}(C)$:

- $\tau = \bot$, or
- $\tau' = \top$, or
- $\mathsf{Con}(\tau) = \mathsf{Con}(\tau')$, or
- τ or τ' is a variable.

Here $\mathsf{Con}(\tau)$ denotes the main constructor of τ. A set, which is not consistent, is called inconsistent.

We now define what it means for a *structural* constraint set to be consistent. A set C is called *ground consistent*, if $b \leq b' \in \mathsf{Cl}(C)$ implies $b \leq_P b'$, for all $b, b' \in P$. A *structural* set C is called *consistent* if and only if it is weakly unifiable and ground consistent. □

Definition 2.3.4 (Substitution) A *substitution* S is a total function mapping type variables to trees, and a substitution is lifted homomorphically to types. The *support* of S, written $\mathrm{Supp}(S)$, is $\{\alpha \in \mathcal{V} \mid \mathcal{S}(\alpha) \neq \alpha\}$, *i.e.*, the set of variables not mapped to themselves by S. If the support of S is finite with $\mathrm{Supp}(S) = \{\alpha_1, \ldots, \alpha_n\}$, then we sometimes write just

$$S = \{\alpha_1 \hookrightarrow S(\alpha_1), \ldots, \alpha_n \hookrightarrow S(\alpha_n)\}$$

If $V \subseteq \mathcal{V}$ then the *restriction of* S *to* V, denoted $S\,|_V$, is the substitution S' such that $S'(\alpha) = S(\alpha)$ for $\alpha \in V$ and $S'(\alpha) = \alpha$ for $\alpha \notin V$. The notation

$$S \oplus \{\alpha_1 \mapsto v_1, \ldots, \alpha_n \mapsto v_n\}$$

refers to the substitution defined by

$$S \oplus \{\alpha_1 \mapsto v_1, \ldots, \alpha_n \mapsto v_n\}(\beta) = \begin{cases} v_i & \text{if } \beta = \alpha_i \\ S(\beta) & \text{otherwise} \end{cases}$$

If $S(\alpha)$ is an *atom* (*i.e.*, a constant in P or a variable) for all α then S is called an *atomic substitution*. If $S(\alpha) \neq S(\beta)$ for all $\alpha, \beta \in V$ with $\alpha \neq \beta$, then S is said to be *injective on* V. If S maps all variables in V to variables and S is injective on V, then S is called a *renaming on* V.

If $\mathbf{t} = C, \Gamma \vdash_P M : \tau$ is a typing judgement, then we let $\mathsf{Var}(C), \mathsf{Var}(\Gamma)$ and $\mathsf{Var}(\tau)$ denote, respectively, the type variables appearing in C, Γ, τ. We let $\mathsf{Var}(\mathbf{t}) = \mathsf{Var}(C) \cup \mathsf{Var}(\Gamma) \cup \mathsf{Var}(\tau)$. If S is a renaming on $\mathsf{Var}(\mathbf{t})$, we sometimes say that S is a renaming on $\mathbf{t}$, for short. For constraint set C we write $S(C) = \{S(\tau) \leq S(\tau') \mid \tau \leq \tau' \in C\}$, and for type assumption set Γ we write $S(\Gamma) = \{x : S(\tau) \mid x : \tau \in \Gamma\}$. We sometimes write the application of a substitution with finite support in reverse order, as in $C\{\alpha_1 \mapsto A_1, \ldots, \alpha_n \mapsto A_n\}$. □

2.3.1 Matching

This section mainly concerns finite structural subtyping.

A distinctive feature of structural as opposed to non-structural subtyping is the first property of Lemma 2.1.1: for two trees to be comparable, they must have the same *shape* (domain.) This entails (as is well known, see [72]) that any satisfiable constraint set must be *weakly unifiable*. As a further consequence of this, a recursive inequality, such as $\alpha \leq \alpha \times \beta$, can only be solved by mapping the recursion variable (α) to an infinite tree in which the recursion is "unwound" into an infinite branch (*e.g.*, $\alpha = \mu\gamma.\gamma \times \beta$ would be a solution for any value of β.)

More precisely, let us say that two trees (or type expressions) t_1 and t_2 C are *matching* if and only if $\mathcal{D}(t_1) = \mathcal{D}(t_2)$ (*i.e.*, t_1 and t_2 have the same shape). A constraint set C is called *matching* if and only if we have τ and τ' matching whenever $\tau \leq \tau' \in C$.

We now turn to the notion of a *most general matching substitution* (see [53]) for *finite, structural* subtyping. Let U_C be the most general (non–circular) unifier of E_C for a weakly unifiable and structural set C. Let $u^C_\alpha = U_C(\alpha)$, and let $\{\alpha_w \mid w \in \mathbf{A}^*\}$ be a set of variables not occurring in C. For each variable α in C define the term θ^C_α with $\mathcal{D}(\theta^C_\alpha) = \mathcal{D}(u^C_\alpha)$ and with

$$\theta^C_\alpha(w) = \begin{cases} u^C_\alpha(w) & \text{if } w \in \mathsf{In}(u^C_\alpha) \\ \alpha_w & \text{if } w \in \mathsf{Lf}(u^C_\alpha) \end{cases}$$

Define the substitution Θ_C from terms to terms by setting

$$\Theta_C(\alpha) = \theta_\alpha^C$$

For the case of *finite, structural* subtyping, a *matching substitution* for a constraint set C is a substitution $S : \mathcal{V} \to T_\Sigma(\mathcal{V})$ such that $S(C)$ is matching. If C is weakly unifiable, then Θ_C is a *most general matching substitution* for C, *i.e.*, $\Theta_C(C)$ is matching, and for any other matching substitution S for C there is a substitution R such that $S = R \circ \Theta_C$ holds on the variables occurring in C.

These properties are standard for *finite*, structural subtyping in the literature (see [53] for the finite case); we shall generalize the notion of matching to the case of structural, recursive subtyping later in this thesis.

Lemma 2.3.5 *(Match Lemma) In the model $\mathcal{T}_\Sigma^{\mathrm{F}}[s]$ one has:*

1. *If $v \models C$, then $v(C)$ is matching.*
2. *If $v \models C$, then $v = v' \circ \Theta_C$ holds on the variables in C for some valuation v'*
3. *$C \models \tau \leq \tau'$ if and only if $\Theta_C(C) \models \Theta_C(\tau) \leq \Theta_C(\tau')$*
4. *If C is an atomic, satisfiable set and $C \models_P \tau \leq \tau'$, then τ and τ' are matching.*

PROOF See Appendix A.1. □

This property holds for both finite and infinite structural subtyping.

Example 2.3.6 (Matching)
Let C be the constraint set

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

Then one has

$\Theta_C(\beta) = \beta_f \times \beta_s$ and

$\Theta_C(\alpha) = (\alpha_{ff} \times \alpha_{fs}) \times (\alpha_{sf} \times \alpha_{ss})$

$\Theta_C(\gamma) = \gamma_\epsilon$

$\Theta_C(\delta) = \delta_\epsilon$

So the set $\Theta_C(C)$ has inequalities

$(\alpha_{ff} \times \alpha_{fs}) \times (\alpha_{sf} \times \alpha_{ss}) \leq (\beta_f \times \beta_s) \times (\beta_f \times \beta_s)$

$\beta_f \times \beta_s \leq \gamma_\epsilon \times \delta_\epsilon$

□

The following example illustrates two important properties of Θ_C:

1. The substitution Θ_C may (even in the absence of type recursion) incur an exponential blow–up in the size of types, but
2. If Θ_C maps a variable to a finite type, then that type must have depth at most linear in the size of C.

Example 2.3.7 (Exponential blow–up)
Let C be a constraint set of the form

$$C = \{\alpha_i \leq \alpha_{i+1} \times \alpha_{i+1}\}_{i=1...n}$$

Then it is easy to verify that the tree $\Theta_C(\alpha_1)$ has an exponential number (exactly 2^n) leaves. However, the tree $\Theta_C(\alpha_1)$ has *depth* of size linear in the size of C, because the depth of Θ_C is no greater than that of $U_C(\alpha_1)$, which can be computed (in the form of a graph representation) in linear time. □

2.3.2 Flattening

This section is again about a basic property of finite, structural systems. It can be generalized to recursive structural systems, and this will be done later.

Lemma 2.3.8 *(Leaf Lemma) Let θ_1, θ_2 be terms with $\mathcal{D}(\theta_1) = \mathcal{D}(\theta_2)$, and let v be a valuation, $v : \mathcal{V} \to T_\Sigma$. Then $v(\theta_1) \leq v(\theta_2)$ holds in T_Σ if and only if $v(\theta_1(w)) \leq^w v(\theta_2(w))$ for every (common) leaf address w in θ_1 and θ_2.*

PROOF The lemma is an easy consequence of Lemma 2.1.1 together with the fact that a valuation preserves constructors homomorphically. □

For a weakly unifiable set C, define the *flattened* set $C^\flat$ by

$$C^\flat = \{\theta(w) \leq^w \theta'(w) \mid \theta \leq \theta' \in \Theta_C(C), w \in \mathsf{Lf}(\theta) \cap \mathsf{Lf}(\theta')\}$$

Observe that, if C is weakly unifiable, then $\mathsf{Lf}(\theta) = \mathsf{Lf}(\theta')$ for all $\theta \leq \theta' \in \Theta_C(C)$. Given constraint set C and variables α, β we define the set of inequalities $[\alpha \leq \beta]_C$ to be the flattened set

$$[\alpha \leq \beta]_C = \{\theta^C_\alpha(w) \leq^w \theta^C_\beta(w) \mid w \in \mathsf{Lf}(\theta^C_\alpha) \cap \mathsf{Lf}(\theta^C_\beta)\}$$

Definition 2.3.9 If C is weakly unifiable, then we say that two variables α and β are *matching in* C if and only if $\Theta_C(\alpha)$ and $\Theta_C(\beta)$ are matching. In this case, $\mathsf{Lf}(\theta^C_\alpha) = \mathsf{Lf}(\theta^C_\beta)$. □

The following lemma is an easy consequence of the defining properties of the most general matching substitution.

Lemma 2.3.10 *(Flattening Lemma) Let C be a weakly unifiable constraint set. Then*

1. *C is satisfiable if and only if $C^\flat$ is satisfiable. More specifically, one has*
   (a) *If $v \models C$, then there is a valuation v' such that $v = v' \circ \Theta_C$ holds on the variables in C, and with $v' \models C^\flat$*
   (b) *$v \models C^\flat$ if and only if $v \circ \Theta_C \models C$*
2. *If α and β are matching in C, then*

$$C \models \alpha \leq \beta \text{ if and only if } C^\flat \models [\alpha \leq \beta]_C$$

PROOF See Appendix A.1 □

2.3.3 Decomposition

The following lemma holds for all systems of subtyping under consideration (structural, non–structural, finite, recursive) in this thesis:

Lemma 2.3.11 *(Decomposition for $\mathcal{T}_\Sigma$) Let t_1, t_2, t_3, t_4 be arbitrary trees in $\mathcal{T}_\Sigma$. Then*

1. *$t_1 \times t_2 \leq t_3 \times t_4$ if and only if $t_1 \leq t_3$ and $t_2 \leq t_4$*
2. *$t_1 \to t_2 \leq t_3 \to t_4$ if and only if $t_3 \leq t_1$ and $t_2 \leq t_4$*

PROOF Direct consequence of the definition of the order $\leq$ on $\mathcal{T}_\Sigma$. □

Lemma 2.3.12 *(Decomposition for $\vdash_P$) Let C be atomic. Then*

1. *$C \vdash_P \tau_1 \times \tau_2 \leq \tau_3 \times \tau_4$ if and only if $C \vdash_P \{\tau_1 \leq \tau_3, \tau_2 \leq \tau_4\}$*
2. *$C \vdash_P \tau_1 \to \tau_2 \leq \tau_3 \to \tau_4$ if and only if $C \vdash_P \{\tau_3 \leq \tau_1, \tau_2 \leq \tau_4\}$.*

PROOF See [53]. □

Notice that the previous lemma does not hold for non–atomic constraint sets in general. For instance, one has $\{\alpha \times \beta \leq \gamma \times \delta\} \vdash_P \alpha \times \beta \leq \gamma \times \delta$, but not $\{\alpha \times \beta \leq \gamma \times \delta\} \vdash_P \alpha \leq \delta$. However, decomposition holds unrestricted for the entailment relation:

Lemma 2.3.13 *(Decomposition for $\models$)*

1. *$C \models \tau_1 \times \tau_2 \leq \tau_3 \times \tau_4$ if and only if $C \models \{\tau_1 \leq \tau_3, \tau_2 \leq \tau_4\}$*
2. *$C \models \tau_1 \to \tau_2 \leq \tau_3 \to \tau_4$ if and only if $C \models \{\tau_3 \leq \tau_1, \tau_2 \leq \tau_4\}$.*

PROOF The implications from right to left are obvious. The implications from left to right follow from Lemma 2.3.11 together with the definition of entailment. □

2.3.4 Substitutivity

Both relations $\vdash_P$ and $\models$ are closed under type substitutions.

Lemma 2.3.14 *(Substitutivity) Let S be a substitution. Then one has*

1. *If $C \vdash_P \tau \leq \tau'$, then $S(C) \vdash_P S(\tau) \leq S(\tau')$*
2. *If $C \models \tau \leq \tau'$, then $S(C) \models S(\tau) \leq S(\tau')$*

PROOF The first property is proven by induction in the proof of $C \vdash_P \tau \leq \tau'$ (see [53]). For the second property, supposing $C \models \tau \leq \tau'$ and $v \models S(C)$, one has $v \circ S \models C$, hence $v \circ S \models \tau \leq \tau'$, hence $v \models S(\tau) \leq S(\tau')$. □

2.4 Typability and satisfiability in partial orders

For the type systems considered in this thesis, the problem of *typability* (does a given term have a type?) is linear time reducible to the problem of *satisfiability* of subtype constraints[1] in an appropriate structure of ordered trees $\mathcal{T}_\Sigma$ [37, 57]. For these systems, the satisfiability problem is the combinatorial bottleneck, and typability is decidable by performing the following steps on a given program M:

1. Extract a linear number of subtype constraints from M, typically extracting one constraint at each node in the syntax tree. Call the resulting constraint set C_M.

2. Test whether C_M is satisfiable in the appropriate structure $\mathcal{T}_\Sigma$.

It matters for the complexity of the satisfiability problem what kind of poset we assume at the base types. In Part II of this thesis, we will assume that the subtype ordering on trees is generated from *lattices*. The significance of this can be seen from previous results on the complexity of subtype satisfiability problems.

An important set of results from previous work establishes that subtype orderings generated from *lattices* have PTIME satisfiability problems, whereas other orderings may generate intractable satisfiability problems. The reason lattices lead to tractable satisfiability problems is that *satisfiability* becomes equivalent to *consistency*. This fundamental property leads to PTIME algorithms for the satisfiability problem, because consistency checking can be reduced to checking properties of the closure of the constraint set, which in turn can be computed via (in some cases, dynamic) transitive closure. Systems for which such results have been shown include atomic subtyping over a lattice of base types, finite structural subtyping with the order on trees generated from a lattice of base types and non–structural recursive subtyping where Σ is a lattice of constructors. On the other hand, it is known that relaxing the structure of the poset Σ may lead to highly complex (intractable) satisfiability problems.

Thus, it appears that lattices are the "right" structure to use, if we wish to construct systems with tractable satisfiability problems [2] Together

[1] In most cases, the typability and satisfiability problems will be polynomial time *equivalent*, [37]

[2] To be precise, it is known that, in many cases, one does not need to require exactly a lattice, but one can do with structures that are in a certain sense sufficiently "lattice–like". See in particular the works of Benke [9, 10].

with the well established fact that lattices are highly useful structures for program analysis, this is our main reason for requiring subtype orders to be generated from lattices in Part II of the thesis.

It will be useful to consider in more depth what is known about the complexity of satisfiability problems, because we will need several results later.

2.4.1 Atomic subtyping

Atomic subtype satisfiability over a poset P is the problem of deciding whether a set C of constraints of the form $A \leq A'$, involving only atomic types, is satisfiable in P. The constants mentioned in C are drawn from P.

The language of atomic inequalities is very inexpressive. There are no operators and no syntactic constructors at all. Nevertheless, it turns out that, already at the level of atomic inequalities, the structure of the poset P is critical for the complexity of the satisfiability problem.

Atomic inequalities over a lattice

If P is a lattice, then one has that satisfiability is equivalent to ground consistency. To explain this in more detail, let C be an atomic constraint set, let α be a variable, and define the sets $\uparrow_C(\alpha)$ and $\downarrow_C(\alpha)$ relative to a given poset P of base types by

$$\uparrow_C(\alpha) = \{b \in P \mid C \vdash_P \alpha \leq b\}$$

$$\downarrow_C(\alpha) = \{b \in P \mid C \vdash_P b \leq \alpha\}$$

Then, as was noted by Tiuryn [71] and by Lincoln and Mitchell [47], one has

Theorem 2.4.1 (Tiuryn [71], Lincoln & Mitchell [47]) *Let C be an atomic constraint set over a lattice L of constants. If C is ground consistent, then*

1. *The valuation $v_\wedge = \{\alpha \mapsto \bigwedge \uparrow_C(\alpha)\}_{\alpha \in \mathit{Var}(C)}$ is a solution to C*
2. *The valuation $v_\vee = \{\alpha \mapsto \bigvee \downarrow_C(\alpha)\}_{\alpha \in \mathit{Var}(C)}$ is a solution to C*

In general, an atomic constraint set over a lattice is satisfiable if and only if it is ground consistent.

Linear time algorithms

To decide satisfiability of atomic subtype inequalities over a finite lattice, one should not proceed in practice by using Theorem 2.4.1 directly, checking ground consistency by computing the closure (which comes down to transitive closure) of the constraint set. Instead, one can regard the satisfiability problem as a fixed–point problem, which can be solved in linear time, for any fixed finite lattice, using essentially Kildall's worklist based fixed point algorithm for data–flow analysis [43]. Satisfiability can be decided in time $h(L) \cdot |C|$, where $h(L)$ is the height of L and $|C|$ is the (textual) size of C. This method works with any monotone functions on L occurring on the left hand side of inequalities of the form

$$f(\alpha_1, \ldots, \alpha_n) \leq A$$

See [66] for an adaptation and a more comprehensive survey of the scope of this method.

To understand the limitations of the linear time framework mentioned above, it is useful to regard inequalities over a lattice as a generalization of propositional Horn–clauses. Horn–clauses emerge in this framework by fixing the lattice L to be the two–point boolean lattice, and by fixing the set of monotone functions to be logical conjunction. Then a Horn–clause $x_1 \wedge \ldots \wedge x_n \Rightarrow y$ is equivalent to the inequality $x_1 \wedge \ldots \wedge x_n \leq y$. One then gets back the linear time Horn–clause decision procedure of Gallier and Dowling [20] as a special case of Kildall's fixed point framework (when the data–flow part is abstracted away, see [66] where this view is developed.) Under this view, it is easy to see, *e.g.*, that adding both $\wedge$ and $\vee$ to the language of inequalities leads to an NP–complete satisfiability problem.

Hard posets

Contrasting with the case of lattices, it is known that there exist very simple looking posets for which the atomic satisfiability problem is NP–complete. In particular, this holds for the so–called *n–crowns* studied by Pratt and Tiuryn in [62]. Since these posets are interesting and we shall use them later in this thesis, let us mention here that the n–crown is the poset with $2n$ elements $0, 1, \ldots, 2n$ ordered by $2i \leq (2i \pm 1) \bmod 2n$. Figure 2.5 shows the simplest interesting crown, the 2–crown. It was shown in [62] that satisfiability of atomic constraints over any n–crown, for $n \geq 2$, is NP–complete. This result was proven as a core part of a more general attempt

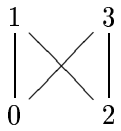


Figure 2.5: A 2-crown

to characterize the structure of posets for which the atomic satisfiability problem is tractable; see also the works of Benke [9, 10].

The conclusion so far is that, even at the level of the weakest possible subtype constraint language, it is reasonable to focus on lattices (or lattice–like structures) of base types, if we want to stay within a tractable framework.

2.4.2 Structural subtyping

The first tractability result for non–atomic subtyping was given for *structural, finite* subtyping over a lattice of type constants, by Tiuryn [71] who shows

Theorem 2.4.2 (Tiuryn [71]) *A constraint set C is satisfiable in $\mathcal{T}^{\mathrm{F}}_{\Sigma}[s]$ if and only if C is consistent (weakly unifiable and ground consistent), when the order on $\mathcal{T}^{\mathrm{F}}_{\Sigma}[s]$ is generated from a lattice. In particular, satisfiability and hence typability with structural, finite subtyping over a lattice is in PTIME.*

Finite, structural satisfiability over a lattice

We briefly consider what is known about the low–level complexity of deciding subtype inequalities over $\mathcal{T}^{\mathrm{F}}_{\Sigma}[s]$, when order is generated from a lattice. The PTIME procedure immediately present in the theorem cited above is as follows:

1. Check for weak unifiability, using unification
2. Compute the closure of the constraint set and check for ground consistency.

The first step is in linear time. It is obvious that the check for ground consistency can be done by computing dynamic transitive closure (DTS) in

a graph representation of the constraint set. However, it is worth noticing that the DTS problem is not inherent for finite structural subtyping, and the *cubic time bottleneck*, which appears hold to for non–structural recursive subtyping [31, 49], does not, in theory at least, characterize the finite structural case. As has been noted before, (see, *e.g.*, [32, 2]), closure in the finite structural case can be computed within subcubic time $M(n)$ required to compute the product of two boolean $n \times n$ matrixes [1]. The reason is that the finite structural order allows a topological stratification of the constraint set, which can be used to control repeated *static* transitive closure of segments of the constraint set; when controlled in this way, these partial static transitive closures can be amortized over the entire constraint set to yield an upper bound of $M(n)$. Intuitively, stratification is based on the shape of $U_C(\tau)$ (Definition 2.3.1) for each subterm τ in the constraint set. Because constraint cycles are not present in a weakly unifiable set over $\mathcal{T}_\Sigma^{\mathrm{F}}[s]$, this leads to a topological ordering, which essentially puts a subterm τ earlier than τ' if $\mathcal{D}(U_C(\tau)) \subseteq \mathcal{D}(U_C(\tau'))$. More details can be found in [32] and in [2]. It is not clear if the sub–cubic result has any practical significance. Transitive closure via boolean matrix multiplication on sparse matrixes appears to be outperformed in practice by dynamic transitive closure algorithms, and constraint graphs are typically sparse.

Finite, structural satisfiability over hard posets

We have so far considered satisfiability in $\mathcal{T}_\Sigma^{\mathrm{F}}[s]$, where we assume that the order is generated from a lattice of base types. The significance of this assumption was also highlighted by Tiuryn in [71], who showed that assuming a non–lattice at the basis of the ordering of trees may lead to PSPACE–hard satisfiability problems. In particular, for each $n \geq 2$, the satisfiability problem in finite trees ordered structurally over any fixed n–crown, is PSPACE-hard (and, by the subsequent result of Frey [27], PSPACE–complete.) Proving PSPACE–hardness is non–trivial; the proof [71] uses techniques from [62] (which was published later.)

Structural, recursive satisfiability

We now consider satisfiability in $\mathcal{T}_\Sigma[s]$, *i.e.*, structural recursive subtyping constraints. The proof of Theorem 2.4.2 for the finite case was obtained by an induction over levels, essentially defined by the topological stratification of the constraint set mentioned earlier. This method is not applicable to

recursive subtype constraints, because cyclic constraints (such as, *e.g.*, $\alpha \leq \alpha \times \beta$) may be present in a solvable constraint set.

Later, Tiuryn and Wand [72] studied the satisfiability problem for structural recursive subtyping over arbitrary posets of base types, using new methods. They show that the satisfiability problem is polynomial time reducible to the problem REG-SAT, satisfiability of *infinite, regular* sets of flat (atomic) inequalities over the poset of base types; the satisfiability problem for such sets was shown to be PSPACE–hard and in DEXPTIME, by a reduction to the emptiness problem for exponentially large Büchi automata. Moreover, their PSPACE–hardness result was shown to hold *for any non-trivial poset of base types* (in particular, including "nice" structures, such as lattices.) They did *not*, however, show that the general, infinitary problem (REG-SAT) is polynomial time *equivalent* to the original problem (structural recursive subtype satisfiability, *i.e.*, satisfiability over infinite trees of *finite* sets of inequalities between *finite* type expressions), and therefore their PSPACE-lower bound does in fact *not* transfer automatically to the original problem.

In fact, it appears that no proof has so far been published for the exact complexity of structural recursive subtype satisfiability over a lattice of base types. Later in this thesis (Chapter 10, Corollary 10.4.3) we show that the problem is in fact in PTIME, again by a reduction of satisfiability to consistency. The resulting algorithm does not bypass the dynamic transitive closure problem, because the unification graph (with respect to E_C) for a satisfiable constraint set may be cyclic, so the stratification method fails in this case; the best upper bound we can give at present is therefore cubic. A sketch of an alternative reduction of the satisfiability problem to the consistency problem can be found in the note [26] by Frey.

2.4.3 Non–structural subtyping

Results of Palsberg and O'Keefe [57] and of Pottier [59], show that satisfiability of *recursive* subtype inequalities over *non-structurally* ordered trees (ordered by the Amadio-Cardelli ordering, as in the present thesis) is in PTIME. Indeed, the problem was again shown to be equivalent to *consistency* of the constraint set, which can be decided in time $O(n^3)$, using dynamic transitive closure (n is the size of the constraint set.) The techniques used to prove this result involves the use of a special kind of automata, which we shall use in this thesis also (we call them constraint automata.)

Theorem 2.4.3 (Palsberg & O'Keefe [57], Pottier [59]) *A constraint set C is satisfiable in $\mathcal{T}_\Sigma[n]$ if and only if C is consistent. In particular, satisfiability is in PTIME.*

By using constraint graphs and automata techniques, Kozen, Palsberg and Schwartzbach [44] have shown that satisfiability in $\mathcal{T}_\Sigma^{\mathrm{F}}[n]$, non–structural finite trees, can also be computed in time $O(n^3)$. In this case also, it is not known how to bypass the dynamic transitive closure problem; stratification methods fail already at the level of finite trees for the non–structural order, because stratification is based on unification (wrt. E_C), and unification may certainly fail for a set satisfiable in $\mathcal{T}_\Sigma^{\mathrm{F}}[n]$; moreover, a satisfiable set may well contain cyclic constraints (consider, *e.g.*, that $\alpha \leq \alpha \times \alpha$ can be solved by mapping α to $\bot$.)

Theorem 2.4.4 (Kozen, Palsberg & Schwartzbach [44]) *Satisfiability of subtype constraints over $\mathcal{T}_\Sigma^{\mathrm{F}}[n]$ is decidable in cubic time.*

2.4.4 Summary

The previous review of subtype satisfiability has shown that the structure of the underlying poset of base types can have a drastic effect on the complexity of subtype satisfiability. We are motivated by these results to make the following decision for Part II of the thesis:

- *We consider only entailment complexity over structures that are generated from lattices of base types*

Our review of the complexity of *lattice*–based subtype satisfiability can be summarized in the following table:

	structural	non–structural
atomic types	$O(n)$	$O(n)$
finite types	$M(n)$	$O(n^3)$
infinite types	$O(n^3)$	$O(n^3)$

By the end of this thesis, we will obtain a similar table for the complexity of the corresponding entailment problems, and we shall be interested in making a comparison. Let us notice already from the outset, though, that there is not a priori any tight relationship between the complexity of satisfiability and the complexity of entailment for the constraints considered in this thesis. The only immediately obvious relationship is that *non*–entailment is just as

hard as satisfiability (– satisfiability of a constraint set C is equivalent to the *non*–entailment $C \not\models \phi$, where ϕ is a formula false in $\mathcal{M}$, and – assuming $\mathcal{M}$ is not the singleton set – such a formula can always be found.) However, for the logically weak constraint language under consideration here, entailment is not necessarily reducible to unsatisfiability, because we do not have a negation sign (or negative constraints, $\not\leq$) in our language, so we cannot use the direct reduction of $C \models \phi$ to unsatisfiability of $C \cup \{\neg\phi\}$, and upper bounds for unsatisfiability cannot be expected to transfer. Indeed, we shall see that this is in fact very far from the case. Still, we shall find that the table above is in some sense reflected in the complexity of entailment.

Part I

The structure and complexity of principal typings

Chapter 3

Minimal typings in atomic subtyping

This chapter shows that equivalence classes of typings, with respect to instantiation, have interesting and useful structure. The instance relation, called $\prec$ below, is an entailment–based version of Fuh and Mishra's syntactic "lazy" instance relation ($\prec_{syn}$, Section 1.5). The type system is atomic.

Our main goal is to characterize *minimal* typings, which, intuitively, are typing judgements with a minimal degree of freedom. Since typings are finite, it is immediate that every term with a typing has a principal typing with a minimal number of distinct variables. It is not obvious, however, how one recognizes that a judgement is minimal in complicated situations, and it is not obvious how typings in an equivalence class are related to minimal typings. We characterize minimal typings and show that they have strong uniqueness properties. This can be understood as a confluence property for simplification.

The results in this chapter will be used in Chapter 5, where we study the size of typings. In that chapter, we will compare the power of several instance relations ($\prec_{syn}$, $\prec$ and a third relation still more powerful than $\prec$) with respect to the simplifications they validate.

The development beginning in the present chapter and continuing through Chapter 5 represents a generalization and extension of our results reported in [65], which were based on the syntactic instance relation $\prec_{syn}$.

We fix a subtyping system, called $\lambda(\models_P)$, to be the *atomic* system (see Section 2.2), which arises by taking the entailment relation in rule [*sub*] of Figure 2.4 to be $\models_P$. We will always assume that the poset P of type

constants is *non–trivial* (*i.e.*, non–discrete), since otherwise inequalities are equivalent to equalities, and no interesting subtype relations can emerge. The relation $C \models_P \tau \leq \tau'$ with C atomic holds if and only if every valuation $v : \mathcal{V} \to P$ satisfying all constraints in C also satisfies the inequality $\tau \leq \tau'$ in finite structural trees, $\mathcal{T}_\Sigma^{\mathrm{F}}[s]$.

3.1 Instance relation

The following definition gives the instance relation called $\prec$. It has the form of the so–called "lazy instance relation" (called $\prec_{syn}$, Section 1.5) defined by Fuh and Mishra in [28], only our relation is more powerful (larger), because we use the model theoretic entailment relation $\models_P$ instead of the syntactic provability relation $\vdash_P$.

Definition 3.1.1 (Instance relation, principal typing)
Let $\mathbf{t}_1 = C_1, \Gamma_1 \vdash_P M : \tau_1, \mathbf{t}_2 = C_2, \Gamma_2 \vdash_P M : \tau_2$ be two atomic judgements, and let S be a type substitution. We say that $\mathbf{t}_2$ *is an instance of* $\mathbf{t}_1$ *under* S, written $\mathbf{t}_1 \prec_S \mathbf{t}_2$, iff

1. $C_2 \models_P S(C_1)$
2. $C_2 \models_P S(\tau_1) \leq \tau_2$
3. $\mathcal{D}(\Gamma_1) \subseteq \mathcal{D}(\Gamma_2)$ and $\forall x \in \mathcal{D}(\Gamma_1).\ C_2 \models_P \Gamma_2(x) \leq S(\Gamma_1(x))$

We say that $\mathbf{t}_2$ *is an instance of* $\mathbf{t}_1$, written $\mathbf{t}_1 \prec \mathbf{t}_2$, iff there exists a type substitution S such that $\mathbf{t}_1 \prec_S \mathbf{t}_2$. We say that $\mathbf{t}_1$ and $\mathbf{t}_2$ are *equivalent*, written $\mathbf{t}_1 \approx \mathbf{t}_2$, iff $\mathbf{t}_1 \prec \mathbf{t}_2$ and $\mathbf{t}_2 \prec \mathbf{t}_1$. Clearly, $\approx$ is an equivalence relation.

A typing judgement for a term M is called a *principal typing for* M if it is derivable in the type system, and it has all other derivable judgements for M as instances. □

If $\mathbf{t} = C, \Gamma \vdash_P M : \tau$ and S is a type substitution, we write $S(\mathbf{t}) = S(C), S(\Gamma) \vdash_P M : S(\tau)$. We say that S is a *renaming on* $\mathbf{t}$ as a shorthand for saying that S is a renaming on $\mathsf{Var}(\mathbf{t})$.

Notice that $\mathbf{t}_1 \prec_S \mathbf{t}_2$ implies $S(\mathbf{t}_1) \prec_{id} \mathbf{t}_2$, and that one always has $\mathbf{t} \prec_S S(\mathbf{t})$.

As was mentioned in the Introduction (Chapter 1), subtyping systems have many different ways of *representing* equivalent typings. This point is

illustrated again in the following simple example [1] from [28].

Example 3.1.2 Let *comp* be the composition combinator defined by

$$comp = \lambda f.\lambda g.\lambda x.f(gx)$$

Then both of the following typings are principal for *comp*:

1. $\mathbf{t}_1$:

$$\{\delta \leq \alpha, \eta \leq \gamma, \beta \leq \upsilon\}, \emptyset \vdash_P comp : (\alpha \to \beta) \to (\gamma \to \delta) \to (\eta \to \upsilon)$$

2. $\mathbf{t}_2$:

$$\emptyset, \emptyset \vdash_P comp : (\alpha \to \beta) \to (\gamma \to \alpha) \to (\gamma \to \beta)$$

□

The remainder of this chapter is organized as follows. We first define what it means for a typing judgement to be *minimal*. We then prove that minimal typings (if they exist) are essentially unique up to renaming substitutions. We then prove that minimal typing judgements always exist. The hard part is to prove existence, because we have by definition built strong uniqueness properties into minimal typings.

3.2 Minimal typings and uniqueness

Kernels

We will need to reason about logical equivalence ($\sim_P$, Section 2.1.8), and in order to do so the following notion is sometimes very useful:

Definition 3.2.1 Let $\mathsf{Th}(P)$ be the set of all inequalities ϕ such that $\models_P \phi$, *i.e.*, the inequalities logically valid in P. The *kernel* [2] of a constraint set C is denoted $\mathsf{Ker}(C)$ and is defined by

$$\mathsf{Ker}(C) = \{\phi \mid C \models_P \phi, \mathsf{Var}(\phi) \subseteq \mathsf{Var}(C), \phi \notin \mathsf{Th}(P)\}$$

That is, $\mathsf{Ker}(C)$ is the set of all inequalities entailed by C that are not logically valid in P and which have all their variables constrained in C. □

[1] The reader should be aware that examples can be scaled up to become *much* more complex. For more examples consult works on subtype simplification, such as [28, 17, 41, 67, 21, 37, 59, 73, 4].

[2] In [73] a notion of constraint kernels is also defined, but it is not the same as ours.

Notice that, since P is finite, $\mathsf{Ker}(C)$ is always a finite set of inequalities. Any constraint set C is logically equivalent to its kernel:

Lemma 3.2.2 *$C \sim_P$ Ker(C)*

PROOF It is obvious that we have $C \models_P \mathsf{Ker}(C)$. To see that $\mathsf{Ker}(C) \models_P C$, suppose that $\phi \in C$. Then $\mathsf{Var}(\phi) \subseteq \mathsf{Var}(C)$, and if $\phi \in \mathsf{Th}(P)$, then $\mathsf{Ker}(C) \models_P \phi$, and otherwise $\phi \in \mathsf{Ker}(C)$. □

Notice that one has

$$\mathsf{Var}(\mathsf{Ker}(C)) \subseteq \mathsf{Var}(C)$$

for any set C. It can occur that C contains more variables than $\mathsf{Ker}(C)$, because C might contain a logically valid inequality, such as $\alpha \leq \top$ (assuming here that P has a top element.) More generally, as our next lemma shows, non–trivial inequalities involving unconstrained variables effectively state that P has a top element or a bottom element. This shows that $\mathsf{Th}(P)$ is not a very interesting set; in fact (as is implied by the lemma below), $\mathsf{Th}(P)$ contains only inequalities of the form $\alpha \leq \alpha$, $b \leq b'$ with $b \leq_P b'$, and in addition to these, in case P has a top element, resp. a bottom element, inequalities $\alpha \leq \top$ and $\bot \leq \alpha$, for all variables α.

Lemma 3.2.3 *Let C be satisfiable.*

1. *If $C \models_P \alpha \leq A$ with $\alpha \neq A$ and $\alpha \notin$ Var(C), then, for some $b \in P$, one has $C \models_P A = b$ and $\models_P \alpha \leq b$. In other words, P must have a top element equivalent to A.*

2. *If $C \models_P A \leq \alpha$ with $\alpha \neq A$ and $\alpha \notin$ Var(C), then, for some $b \in P$, one has $C \models_P A = b$ and $\models_P b \leq \alpha$. In other words, P must have a bottom element equivalent to A.*

PROOF See Appendix A.2 □

Specialization and minimality

In order to define minimality we first need the following notion of *specialization.*

Definition 3.2.4 (Specialization)
Let $\mathbf{t}_1 = C_1, \Gamma_1 \vdash_P M : \tau_1$ and $\mathbf{t}_2 = C_2, \Gamma_2 \vdash_P M : \tau_2$. We say that $\mathbf{t}_1$ *is a specialization of* $\mathbf{t}_2$, written $\mathbf{t}_1 \ll \mathbf{t}_2$, iff there exists an atomic substitution S such that

1. $C_1 \subseteq \mathsf{Ker}(S(C_2))$
2. $\tau_1 = S(\tau_2)$
3. $\mathcal{D}(\Gamma_1) \subseteq \mathcal{D}(\Gamma_2)$ and $\Gamma_1(x) = S(\Gamma_2(x))$ for all $x \in \mathcal{D}(\Gamma_1)$.

We may write $\mathbf{t}_1 \ll_S \mathbf{t}_2$ to signify that $\mathbf{t}_1 \ll \mathbf{t}_2$ under substitution S. □

Notice that, if $\mathbf{t} \ll_S \mathbf{t}'$, then $\mathbf{t} \prec_{id} S(\mathbf{t}')$.

Example 3.2.5 Consider the typings $\mathbf{t}_1$ and $\mathbf{t}_2$ shown in Example 3.1.2 and take $S = \{\delta \mapsto \alpha, \eta \mapsto \gamma, \upsilon \mapsto \beta\}$. Then $\mathbf{t}_2 \ll_S \mathbf{t}_1$. □

Lemma 3.2.6 *The relation* $\ll$ *is transitive: if* $\mathbf{t}_1 \ll_{S_1} \mathbf{t}_2$ *and* $\mathbf{t}_2 \ll_{S_2} \mathbf{t}_3$, *then* $\mathbf{t}_1 \ll_{S_1 \circ S_2} \mathbf{t}_3$.

PROOF See Appendix A.2 □

We restrict attention to derivable judgements with satisfiable constraint sets, since only such judgements represent well–typings. For a derivable typing judgement $\mathbf{t}$ with satisfiable constraint set we let $[\mathbf{t}]$ denote the equivalence class of $\mathbf{t}$ with respect to $\approx$, *i.e.*, $[\mathbf{t}] = \{\mathbf{t}' \mid \mathbf{t}' \approx \mathbf{t}\}$.

We now define what it means for a typing judgement to be *minimal*. Intuitively, a minimal typing $\mathbf{t}$ is a *most specialized* element within $[\mathbf{t}]$. It is therefore, in a certain sense, an irredundant representative in its equivalence class, where the typing information has been made as explicit as possible.

Definition 3.2.7 (Minimality)
A typing judgement $\mathbf{t}$ is called *minimal* iff it holds for all $\mathbf{t}' \in [\mathbf{t}]$ that $\mathbf{t} \ll \mathbf{t}'$. □

Notice that it can be quite non-trivial to establish that a given typing is minimal, because the definition of minimality quantifies over all the infinitely many members of an equivalence class, which may be related in non–trivial ways. In order to reason about minimality it will therefore be necessary to develop some characterizations of this notion. We shall do so in Section 4.2 below (Theorem 4.2.5).

Uniqueness

The relation $\ll$ is a partial order up to renaming substitutions and "taking the kernel" of constraint sets:

Theorem 3.2.8 *(Uniqueness of minimal typings)*
If $\mathbf{t}_1 \ll_{S_2} \mathbf{t}_2$ *and* $\mathbf{t}_2 \ll_{S_1} \mathbf{t}_1$ *then* S_i *is a renaming on* $\mathbf{t}_i$, $i = 1, 2$.

PROOF See Appendix A.2 □

3.3 Fully substituted typings

We will show that minimal typings can be found in every equivalence class $[\mathbf{t}]$, essentially by taking a *full substitution instance* (see definition below) of $\mathbf{t}$ within $[\mathbf{t}]$. Minimality will follow from a strong uniqueness property of fully substituted judgements within each equivalence class $[\mathbf{t}]$. Uniqueness of fully substituted typings can be regarded as a *confluence* property for simplification transformations that work by applying substitutions to typings (recall Section 1.5 where this idea was introduced.) We generally assume that constraint sets in typing judgements are satisfiable, since we have no interest in other typings.

Definition 3.3.1 (Full substitution instance)
If $\mathbf{t}' = S(\mathbf{t})$, then we say that $\mathbf{t}'$ is a *substitution instance of* $\mathbf{t}$ *under* S. A typing judgement $\mathbf{t}$ is called *fully substituted* iff, whenever $S(\mathbf{t}) \in [\mathbf{t}]$, then S is a renaming on $\mathsf{Var}(\mathbf{t})$. A *full substitution instance* of a typing $\mathbf{t}$ is a substitution instance of $\mathbf{t}$ which is fully substituted. □

Notice that $\mathbf{t}' = S(\mathbf{t})$ implies $\mathbf{t}' \ll_S \mathbf{t}$ but not vice versa. Any substitution instance $S(\mathbf{t})$ of a typing $\mathbf{t}$ is an instance of $\mathbf{t}$, provided S is an atomic substitution such that $S(C)$ is satisfiable, since in this case $S(\mathbf{t})$ is an atomic judgement with $\mathbf{t} \prec_S S(\mathbf{t})$. Hence, if $S(\mathbf{t}) \prec \mathbf{t}$, then $S(\mathbf{t}) \approx \mathbf{t}$.

Lemma 3.3.2 *Let* $\mathbf{t}$ *be an atomic judgement. If* $S(\mathbf{t}) \in [\mathbf{t}]$, *then* $S \mid_{\mathsf{Var}(\mathbf{t})}$ *is an atomic substitution.*

PROOF Let $\mathbf{t} = C, \Gamma \vdash_P M : \tau$. By the assumptions, $\mathbf{t}' = S(\mathbf{t})$ is an atomic judgement, so $S \mid_{\mathsf{Var}(C)}$ must be an atomic substitution. Since $\mathbf{t}' \in [\mathbf{t}]$, we have $\mathbf{t} \approx \mathbf{t}'$, which entails, via the Match Lemma (Lemma 2.3.5) together with the definition of $\approx$, that $S(\tau)$ matches τ and that $\Gamma(x)$ matches $S(\Gamma(x))$

for all $x \in \mathcal{D}(\Gamma)$. It follows that $S\,|_{\mathsf{Var}(\tau)\cup\mathsf{Var}(\Gamma)}$ is an atomic substitution. We have now shown that $S\,|_{\mathsf{Var}(\mathbf{t})}$ is atomic. □

By repeated application of non-renaming atomic substitutions to a typing, Lemma 3.3.2 entails that for any typing judgement $\mathbf{t}$, there exists a full substitution instance of $\mathbf{t}$ within $[\mathbf{t}]$. Hence, every atomic typing has a substitution instance which is fully substituted. We will now study how any two equivalent, fully substituted typings are related. To this end the following lemma is useful.

Lemma 3.3.3 *If* $\mathbf{t}_1 \prec_{S_1} \mathbf{t}_2$ *and* $\mathbf{t}_2 \prec_{S_2} \mathbf{t}_1$*, then* $S_1(\mathbf{t}_1) \prec_{S_2} \mathbf{t}_1$ *and* $S_2(\mathbf{t}_2) \prec_{S_1} \mathbf{t}_2$.

PROOF See Appendix A.2 □

Definition 3.3.4 If $\mathbf{t}_1 \prec_{S_1} \mathbf{t}_2$ and $\mathbf{t}_2 \prec_{S_2} \mathbf{t}_1$ where S_1 is a renaming on $\mathsf{Var}(\mathbf{t}_1)$ and S_2 is a renaming on $\mathsf{Var}(\mathbf{t}_2)$, then we say that $\mathbf{t}_1$ and $\mathbf{t}_2$ are *equivalent under renaming* and we write $\mathbf{t}_1 \approx^\bullet \mathbf{t}_2$ in this case. □

Lemma 3.3.3 leads to:

Lemma 3.3.5 *If* $\mathbf{t}_1 \approx \mathbf{t}_2$ *and* $\mathbf{t}_1$ *and* $\mathbf{t}_2$ *are fully substituted, then* $\mathbf{t}_1 \approx^\bullet \mathbf{t}_2$.

PROOF By $\mathbf{t}_1 \approx \mathbf{t}_2$ we have $\mathbf{t}_1 \prec_{S_1} \mathbf{t}_2$ and $\mathbf{t}_2 \prec_{S_2} \mathbf{t}_1$ for some substitutions S_1, S_2. By Lemma 3.3.3 we then have $S_1(\mathbf{t}_1) \prec_{S_2} \mathbf{t}_1$ and $S_2(\mathbf{t}_2) \prec_{S_1} \mathbf{t}_2$, hence $S_1(\mathbf{t}_1) \in [\mathbf{t}_1]$ and $S_2(\mathbf{t}_2) \in [\mathbf{t}_2]$. Since $\mathbf{t}_1$ is fully substituted, it follows that S_1 is a renaming on $\mathsf{Var}(\mathbf{t}_1)$ and since $\mathbf{t}_2$ is fully substituted, it follows that S_2 is a renaming on $\mathsf{Var}(\mathbf{t}_2)$. Then $\mathbf{t}_1 \prec_{S_1} \mathbf{t}_2$ and $\mathbf{t}_2 \prec_{S_2} \mathbf{t}_1$ establish that $\mathbf{t}_1 \approx^\bullet \mathbf{t}_2$. □

3.4 Acyclic constraints

In order to show that fully substituted typings lead to minimal typings, we need to strengthen the conclusion of Lemma 3.3.5. Before doing so, we must consider *cycle elimination* in constraint sets.[3]

[3] Cycle elimination is perhaps the pragmatically single most important optimization of constraint sets in constraint based program analysis; see [23] for a recent contribution.

Definition 3.4.1 An atomic constraint set C is called *cyclic*, if and only if C entails a non–trivial equation, *i.e.*, there are atoms A and A' with $A \neq A'$ such that $C \models_P A = A'$. A constraint set, which is not cyclic, is called *acyclic*. □

We generally assume here that constraint sets are satisfiable. For such sets, only cycles involving a variable (*i.e.*, where either A or A' is a variable in the definition above) are possible, and these are the only interesting cases here. The following lemma says that it is sound to eliminate cycles in a typing:

Lemma 3.4.2 *(Cycle elimination) Let* $\mathbf{t} = C, \Gamma \vdash_P M : \tau$ *be any atomic judgement.*

1. *If* $C \models_P \alpha = A$, *then* $S(\mathbf{t}) \approx \mathbf{t}$, *with* $S = \{\alpha \mapsto A\}$.
2. *There is a substitution instance* $\mathbf{t}'$ *of* $\mathbf{t}$ *such that* $\mathbf{t}'$ *has an acyclic constraint set and with* $\mathbf{t}' \approx \mathbf{t}$.
3. *If* $\mathbf{t}$ *is fully substituted with* C *satisfiable, then* C *is acyclic.*

PROOF See Appendix A.2 □

If a constraint set C is acyclic and satisfiable, then it can only entail relations between variables that are actually constrained in C:

Lemma 3.4.3 *Let* C *be satisfiable. If* C *is acyclic and* $\alpha \neq \beta$, *then* $C \models_P \alpha \leq \beta$ *implies* $\{\alpha, \beta\} \subseteq \mathsf{Var}(C)$.

PROOF Suppose that $C \models_P \alpha \leq \beta$. If $\alpha \notin \mathsf{Var}(C)$, then Lemma 3.2.3 shows that we would have $C \models \beta = b$ with $\models_P \alpha \leq b$, for some $b \in P$. Then C would be cyclic, so $\alpha \in \mathsf{Var}(C)$ must be the case, because C is acyclic. A similar argument establishes that $\beta \in \mathsf{Var}(C)$. □

Lemma 3.4.4 *Let* C_1 *and* C_2 *be atomic constraint sets.*

1. *If* $\mathsf{Ker}(C_1) \subseteq \mathsf{Ker}(C_2)$ *then* $C_2 \models_P C_1$
2. *If* C_1 *is acyclic and satisfiable, then* $C_1 \models_P C_2$ *implies* $\mathsf{Ker}(C_2) \subseteq \mathsf{Ker}(C_1)$
3. *If* C_1 *and* C_2 *are both acyclic and satisfiable, then* $C_1 \sim_P C_2$ *if and only if* $\mathsf{Ker}(C_1) = \mathsf{Ker}(C_2)$

PROOF See Appendix A.2 □

Lemma 3.4.5 *Let C_1, C_2 be atomic constraint sets, both of which are acyclic and satisfiable. Assume that $C_1 \models_P S_2(C_2)$ and $C_2 \models_P S_1(C_1)$ where S_i is a renaming on $\mathsf{Var}(C_i)$, $i = 1, 2$. Then $C_1 \sim_P S_2(C_2)$ and $C_2 \sim_P S_1(C_1)$.*

PROOF See Appendix A.2 □

Lemma 3.4.6 *(Anti–symmetry) Let C be atomic and satisfiable. If C is acyclic, then $C \models_P \tau \leq \tau'$ and $C \models_P \tau' \leq \tau$ imply $\tau = \tau'$.*

PROOF See Appendix A.2 □

3.5 Existence of minimal typings

We continue to strengthen the uniqueness properties of fully substituted typings. After that, we can show how minimal typings can be obtained. A main technical lemma follows first:

Lemma 3.5.1 *Let S be a substitution and C an atomic, satisfiable constraint set with variable $\alpha \in \mathsf{Var}(C)$. Assume*

(i) *S is a renaming on $\mathsf{Var}(C)$*

(ii) *$C \models_P S(C)$*

(iii) *C is acyclic*

(iv) *either $C \models_P S(\alpha) \leq \alpha$ or $C \models_P \alpha \leq S(\alpha)$*

Then $S(\alpha) = \alpha$.

PROOF See Appendix A.2 □

The following proposition is the central technical result in this section. The tricky part is to prove properties *(ii)* and *(iii)* of the proposition.

Proposition 3.5.2 *Let $\mathbf{t}_1 = C_1, \Gamma_1 \vdash_P M : \tau_1$ and $\mathbf{t}_2 = C_2, \Gamma_2 \vdash_P M : \tau_2$ be judgements with satisfiable constraint sets. If $\mathbf{t}_1 \approx^\bullet \mathbf{t}_2$ and $\mathbf{t}_1, \mathbf{t}_2$ are both fully substituted, then there is a renaming S on $\mathbf{t}_2$ such that*

(*i*) $C_1 \sim_P S(C_2)$

(*ii*) $\tau_1 = S(\tau_2)$

(*iii*) $\mathcal{D}(\Gamma_1) = \mathcal{D}(\Gamma_2)$ *and* $\Gamma_1(x) = S(\Gamma_2(x))$ *for all* $x \in \mathcal{D}(\Gamma_1)$.

PROOF Since the judgements are fully substituted with satisfiable constraint sets, we know from Lemma 3.4.2 that the sets C_1 and C_2 must be acyclic. By $t_1 \approx^\bullet t_2$, we know that $t_1 \prec_{S_1} t_2$ and $t_2 \prec_{S_2} t_1$ with S_1 a renaming on t_1 and S_2 a renaming on t_2.

We claim that the proposition holds with $S = S_2$. Part (*i*) follows easily from Lemma 3.4.5 (applicable, since the constraint sets are satisfiable and acyclic) together with the assumptions, which yield

$$C_1 \models_P S_2(C_2) \tag{3.1}$$

and

$$C_2 \models_P S_1(C_1) \tag{3.2}$$

Part (*ii*) and (*iii*) are similar to each other, so we consider only part (*ii*).

To see that (*ii*) holds with $S = S_2$, we first record that, by our assumptions, we have

$$C_1 \models_P S_2(\tau_2) \leq \tau_1 \tag{3.3}$$

and

$$C_2 \models_P S_1(\tau_1) \leq \tau_2 \tag{3.4}$$

Now, we know from Lemma 3.3.3 that $S_2(S_1(\mathbf{t}_1)) \prec_{id} \mathbf{t}_1$, hence we have

$$S_2(S_1(\mathbf{t}_1)) \approx \mathbf{t}_1 \tag{3.5}$$

We therefore have

$$C_1 \models_P S_2(S_1(\tau_1)) \leq \tau_1 \tag{3.6}$$

and, together with the assumption that $\mathbf{t}_1$ is fully substituted, (3.5) shows that

$$S_2 \circ S_1 \text{ is a renaming on } \mathsf{Var}(\mathbf{t}_1) \tag{3.7}$$

Moreover, by $S_2(S_1(\mathbf{t}_1)) \prec_{id} \mathbf{t}_1$, we also have

$$C_1 \models_P S_2(S_1(C_1)) \tag{3.8}$$

We now *claim* that in fact we can strengthen (3.6) to

$$(*) \quad S_2(S_1(\tau_1)) = \tau_1$$

To see that $(*)$ is true, assume that

$$S_2(S_1(\tau_1)) \neq \tau_1 \tag{3.9}$$

aiming for contradiction. Then there is a variable α occurring in τ_1 such that

$$(S_2 \circ S_1)(\alpha) \neq \alpha \tag{3.10}$$

Because (3.7) $S_2 \circ S_1$ is a renaming on $\mathbf{t}_1$ and $\alpha \in \mathsf{Var}(\tau_1) \subseteq \mathsf{Var}(\mathbf{t}_1)$, it must be the case that $S_2 \circ S_1$ is a variable, so we can set $S_2 \circ S_1(\alpha) = \beta$ for some variable β. Now, because of (3.6), the Decomposition Lemma entails that α and β must be comparable under C_1, *i.e.*,

$$\text{either } C_1 \models_P S_2(S_1(\alpha)) \leq \alpha \text{ or } C_1 \models_P \alpha \leq S_2(S_1(\alpha)) \tag{3.11}$$

Since α and β are variables and (3.10) $\beta \neq \alpha$, we must have

$$\alpha \in \mathsf{Var}(C_1) \tag{3.12}$$

since otherwise α and β could not be comparable under the hypotheses C_1, according to Lemma 3.4.3; this lemma is applicable, because C_1 is satisfiable and

$$C_1 \text{ is acyclic} \tag{3.13}$$

which, in turn, is true by Lemma 3.4.2, because $\mathbf{t}_1$ is assumed to be fully substituted. Now, (3.7), (3.8), (3.11), (3.12) and (3.13) allow us to apply Lemma 3.5.1 to the substitution $S_2 \circ S_1$ on C_1, and the Lemma shows that we must have $S_2(S_1(\alpha)) = \alpha$. This contradicts (3.10), and so we must reject the assumption (3.9), and $(*)$ is thereby established.

Now, by $(*)$ together with part (i) of the present lemma, established above, we have $C_1 \sim_P S_2(C_2)$. By the assumptions, which yield $C_2 \models_P S_1(\tau_1) \leq \tau_2$, we then get, via the Substitution Lemma using the substitution S_2, that

$$C_1 \models_P S_2(S_1(\tau_1)) \leq S_2(\tau_2)$$

Applying the equation $(*)$ to this relation, we get

$$C_1 \models_P \tau_1 \leq S_2(\tau_2) \tag{3.14}$$

We know (3.13) that C_1 is acyclic, and we have (3.3) that $C_1 \models_P S_2(\tau_2) \leq \tau_1$; together with (3.14), this shows (using Lemma 3.4.6 and C_1 acyclic) that, in fact, $\tau_1 = S_2(\tau_2)$, as desired. We have now shown property (ii) of the proposition. Property (iii) follows by the same reasoning. □

Let $\mathsf{Red}(C) = C \setminus \mathsf{Th}(P)$. Then $\mathsf{Red}(C) = C \cap \mathsf{Ker}(C)$, and we have $\mathsf{Red}(C) \subseteq \mathsf{Ker}(C)$. We are now ready to prove:

Theorem 3.5.3 *(Existence of minimal typings)*
Let $\mathbf{t} = C, \Gamma \vdash_P M : \tau$ *be any atomic, judgement with* C *satisfiable, and let* $S(\mathbf{t})$ *be a full substitution instance of* $\mathbf{t}$ *within* $[\mathbf{t}]$. *Define* $\tilde{\mathbf{t}}$ *by*

$$\tilde{\mathbf{t}} = \mathit{Red}(S(C)), S(\Gamma) \vdash_P M : S(\tau)$$

Then $\tilde{\mathbf{t}}$ *is a minimal judgement in* $[\mathbf{t}]$.

PROOF Clearly, $\tilde{\mathbf{t}}$ is fully substituted, and moreover $\tilde{\mathbf{t}} \approx \mathbf{t}$ because one has $\mathsf{Red}(S(C)) \sim_P S(C)$. Now let $\mathbf{t}' = C', \Gamma' \vdash_P M : \tau'$ be an arbitrary judgement in $[\mathbf{t}]$ with C' satisfiable. We must show that $\tilde{\mathbf{t}} \ll \mathbf{t}'$. Let $S'(\mathbf{t}')$ be a full substitution instance of $\mathbf{t}'$ within $[\mathbf{t}'] = [\mathbf{t}]$. Then $S(\mathbf{t}) \approx S'(\mathbf{t}')$, and therefore Lemma 3.3.5 and Proposition 3.5.2 together imply that there exists a renaming R on $S'(\mathbf{t}')$ such that

(*i*) $S(C) \sim_P R(S'(C'))$

(*ii*) $S(\tau) = R(S'(\tau'))$

(*iii*) $\mathcal{D}(\Gamma) = \mathcal{D}(\Gamma')$ and $S(\Gamma(x)) = R(S'(\Gamma'(x)))$ for all $x \in \mathcal{D}(\Gamma)$.

Since $S(C)$ and $R(S'(C'))$ must both be acyclic (by Lemma 3.4.2), it follows from (*i*) and Lemma 3.4.4 (part 3) that

$$\mathsf{Ker}(S(C)) = \mathsf{Ker}(R(S'(C'))$$

But $\mathsf{Red}(S(C)) \subseteq \mathsf{Ker}(S(C))$, and together with (*ii*) and (*iii*) this shows that

$$\tilde{\mathbf{t}} \ll_{R \circ S'} \mathbf{t}'$$

thereby proving the theorem. □

3.6 Minimality and the size of judgements

Minimal typings are intended to be optimized presentations of equivalent typings. If $\mathbf{t} = C, \Gamma \vdash_P M : \tau$ is a minimal judgement, then the types τ and $\Gamma(x)$ for $x \in \mathcal{D}(\Gamma)$ are optimal, in the sense that these types are as specific as possible within the equivalence class $[\mathbf{t}]$. We may say that these types are

irredundant, because they exhibit a minimal degree of freedom: whenever they distinguish between two type variables, it is logically necessary to do so, on pain of losing typing power (falling out of the equivalence class). We have the following important property, stated in the proposition below. It shows that, with respect to "degree of freedom" as measured by the number of variables distinguished, our minimal typings are optimal. This property will be a key element in our lower bound proof for the size of principal typings in Chapter 5.

Proposition 3.6.1 *If* $\mathbf{t} = C, \Gamma \vdash_P M : \tau$ *is minimal and* $\mathbf{t}' = C', \Gamma' \vdash_P M : \tau'$ *is any typing judgement such that* $\mathbf{t} \approx \mathbf{t}'$*, then*

1. $|\mathit{Var}(C)| \leq |\mathit{Var}(C')|$
2. $|\mathit{Var}(\tau)| \leq |\mathit{Var}(\tau')|$
3. $|\mathit{Var}(\Gamma)| \leq |\mathit{Var}(\Gamma')|$

PROOF Because $\mathbf{t}$ is minimal, we have $\mathbf{t} \ll_S \mathbf{t}'$ for some substitution S. By the definition of $\ll_S$, this means that

(i) $C \subseteq \mathsf{Ker}(S(C'))$

(ii) $\tau = S(\tau')$

(iii) $\mathcal{D}(\Gamma) \subseteq \mathcal{D}(\Gamma')$ and $\Gamma(x) = S(\Gamma'(x))$ for all $x \in \mathcal{D}(\Gamma)$

The last two claims of the proposition are clearly implied by (ii) and (iii), respectively. To see that the first claim is also true, (i) shows that we have

$$\mathsf{Var}(C) \subseteq \mathsf{Var}(\mathsf{Ker}(S(C'))) \subseteq \mathsf{Var}(S(C'))$$

and therefore $|\mathsf{Var}(C)| \leq |\mathsf{Var}(S(C'))|$; on the other hand, we evidently have $|\mathsf{Var}(S(C'))| \leq |\mathsf{Var}(C')|$. In total, $|\mathsf{Var}(C)| \leq |\mathsf{Var}(C')|$. □

Using this proposition, we can prove lower bounds on the number of variables present in *all* principal typings for a given term by proving corresponding lower bounds for *minimal* principal typings only.

Chapter 4

Minimization in atomic subtyping

In this chapter, we will study two typing transformations, called G and S; these will be entailment–based versions of corresponding transformations defined by Fuh and Mishra [28], which were based on the relation $\vdash_P$[1]

We studied the transformations of [28] in [65]. There are two reasons why we study the transformations here. *Firstly*, Theorem 4.2.5, the main result in this chapter, shows that S–simplification is partially complete, *i.e.*, it computes minimal typings, provided certain conditions are satisfied. Since S–simplification is a relatively simple transformation, Theorem 4.2.5 thereby singles out specific conditions under which it is easy to recognize what a minimal typing looks like. This result is used in the lower bound proof for $\lambda_{\leq}(\models_P)$ in Chapter 5. *Secondly*, the G- and S–simplification are important transformations in their own right, and they will be used to simplify typings in Chapter 5. If one looks at the literature on subtype simplification, one will find that a very significant subset of all transformations suggested for practical use can be understood in terms of S and G. A possible explanation for this is that these transformations apparently capture most of the interesting PTIME–computable simplifications we can come up with. In contrast, we show (Theorem 4.1.5 below) that, unless P = NP, there can be no polynomial time procedure for computing minimal typings, *no matter* what the structure of P is.

[1] I do not know what the names G and S, used by Fuh and Mishra, stand for or are intended to suggest; perhaps "general" and "special", respectively.

4.1 G–simplification and G–minimization

We need a few definitions in order to introduce the G- and S–transformations.

Definition 4.1.1 (Observable types and internal variables) Given typing $\mathbf{t} = C, \Gamma \vdash_P M : \tau$, let the *observable types* in $\mathbf{t}$, denoted $\mathsf{Obv}(\mathbf{t})$, be the constants in P and type variables appearing in Γ or τ, *i.e.*, $\mathsf{Obv}(\mathbf{t}) = \mathsf{Var}(\Gamma) \cup \mathsf{Var}(\tau) \cup P$. Let the *internal variables* in $\mathbf{t}$, denoted $\mathsf{Intv}(\mathbf{t})$, be the set $\mathsf{Intv}(\mathbf{t}) = \mathsf{Var}(C) \setminus \mathsf{Obv}(\mathbf{t})$. □

The G–transformation changes only the internal variables of a typing, whereas S–transformation changes only the observable variables of a typing.

There are two kinds of G-transformation, one stronger than the other. The weaker one is called *G–simplification*, the stronger one is called *G–minimization*. They both eliminate *internal* variables (and only such) from constraint sets. To define the transformations, first define the functions $\Uparrow_C$ and $\Downarrow_C$ by[2]

$$\Uparrow_C(A) = \{A' \mid C \models_P A \leq A'\}$$

and

$$\Downarrow_C(A) = \{A' \mid C \models_P A' \leq A\}$$

We use these functions to define

Definition 4.1.2 (G–subsumption, G–simplification) For a variable $\alpha \in \mathsf{Var}(C)$ and an atom A, we say that α is *G-subsumed by A with respect to C*, written $\alpha \sqsubseteq_{\mathsf{g}} A$ (where C is understood), if and only if we have

1. $\Uparrow_C(\alpha) \setminus \{\alpha\} \subseteq \Uparrow_C(A)$, and
2. $\Downarrow_C(\alpha) \setminus \{\alpha\} \subseteq \Downarrow_C(A)$

The transformation $\mapsto_{\mathsf{g}}$ on typings, called *G–simplification*, is then defined as follows. Let $\mathbf{t} = C, \Gamma \vdash_P M : \tau$. Then $\mathbf{t} \mapsto_{\mathsf{g}} C\{\alpha \mapsto A\}, \Gamma \vdash_P M : \tau$ if and only if we have

[2]The sets $\Uparrow_C$ and $\Downarrow_C$ correspond to the sets $\uparrow_C$ and $\downarrow_C$, introduced in Section 2.4.1, the former using $\models_P$ instead of $\vdash_P$ in their definition. The G- and S–transformations to be introduced here are variations on corresponding transformation defined by Fuh and Mishra [28] and studied by the author in [65], with the only difference that we now use the sets $\Uparrow_C$ and $\Downarrow_C$ instead of $\uparrow_C$ and $\downarrow_C$.

1. $\alpha \in \mathsf{Intv}(C)$, and
2. $\alpha \neq A$, and
3. $\alpha \sqsubseteq_{\mathsf{G}} A$ with respect to C.

□

Note that, if $\mathbf{t} = C, \Gamma \vdash_P M : \tau$ and $\mathbf{t}' = C\{\alpha \mapsto A\}, \Gamma' \vdash_P M : \tau'$ with $\mathbf{t} \mapsto_{\mathsf{G}} \mathbf{t}'$, one has $\Gamma = \Gamma'$ and $\tau = \tau'$. Moreover, G–simplification is sound, because we have $\mathbf{t} \approx \mathbf{t}'$, whenever $\mathbf{t} \mapsto_{\mathsf{G}} \mathbf{t}'$. To see this, we first show that $C \models_P C\{\alpha \mapsto A\}$. So suppose that $\phi \in C\{\alpha \mapsto A\}$. Then w.l.o.g. $\phi = A \leq A'$ with $\alpha \leq A' \in C$ (the case $\phi = A' \leq A$ is similar), and therefore $A' \in \Uparrow_C(\alpha)$, hence (by $\alpha \sqsubseteq_{\mathsf{G}} A$) we have $A' \in \Uparrow_C(A)$ and so $C \models_P A \leq A'$, which shows that $\mathbf{t}' \prec_{id} \mathbf{t}$. On the other hand, we evidently have $\mathbf{t} \prec_S \mathbf{t}'$ with $S = \{\alpha \mapsto A\}$.

An important property of G–simplification is that it may be efficiently computable. This depends on what P is. For instance, if P is an n–crown, $n \geq 2$, then clearly the problem $C \models_P \alpha \leq \beta$ is coNP–complete (by NP–completeness of the corresponding satisfiability problem, see Section 2.4.1), and in such cases, the transformation will be intractable. However, we will show later in this thesis that, if P is a lattice, then the atomic entailment problem $C \models_P \alpha \leq \beta$ can be computed in linear time. Moreover, the simplifications defined by Fuh and Mishra (using $\vdash_P$ instead of $\models_P$) are computable in PTIME regardless of what P is. These transformations can be considered as natural approximations to the corresponding simplifications based on $\models_P$.

G–simplification can be regarded as an approximation to a more powerful transformation, which we will denote $\mapsto_{\mathsf{G}^+}$, called *G-minimization*. It is defined as follows. Given typing $\mathbf{t} = C, \Gamma \vdash_P M : \tau$, let $\mathsf{Subs}(\mathbf{t})$ be the set of atomic substitutions S such that

$$S(\alpha) = \begin{cases} \alpha & \text{if } \alpha \notin \mathsf{Intv}(\mathbf{t}) \\ A \in \mathsf{Var}(C) \cup P & \text{if } \alpha \in \mathsf{Intv}(\mathbf{t}) \end{cases}$$

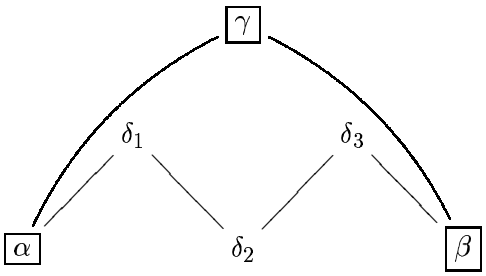
Then $\mathbf{t}_1 \mapsto_{\mathsf{G}^+} \mathbf{t}_2$ iff there exists $S \in \mathsf{Subs}(\mathbf{t}_1)$ such that $C_2 = S(C_1)$, S is not a renaming on $\mathsf{Var}(C_1)$ and $C_1 \models_P S(C_1)$.

Observe that both $\mapsto_{\mathsf{G}}$ and $\mapsto_{\mathsf{G}^+}$ are terminating, since the size of $\mathsf{Intv}(t)$ shrinks at every reduction step.

The following example shows why $\mapsto_{\mathsf{G}}$ is an incomplete approximation to $\mapsto_{\mathsf{G}^+}$. The example shows that, in order to G-minimize, it is not sufficient

to consider substitutions with singleton support[3], and this explains, in part, why G-minimization is hard to compute (Theorem 4.1.5.)

Example 4.1.3 Consider constraint set C below, where observable variables are shown inside a box:



Call an atomic substitution S *simple* if $\mathsf{Supp}(S)$ is a singleton set. It can be seen that C is *G-simplified* (*i.e.*, in normal form with respect to $\mapsto_{\mathsf{g}}$), because it is not possible to find a simple, non–identity substitution S of the form $\{\delta_i \mapsto A\}$, $A \in \{\alpha, \beta, \gamma, \delta_1, \delta_2, \delta_3\}$, such that $C \models_P S(C)$. However, the set C is *not G-minimal*, because the substitution

$$S_{min} = \{\delta_1 \mapsto \gamma, \delta_2 \mapsto \gamma, \delta_3 \mapsto \gamma\}$$

satisfies $C \models_P S_{min}(C)$. Note that even though S_{min} can be factored into a sequence of simple substitutions, as $S_{min} = \{\delta_1 \mapsto \gamma\} \circ \{\delta_2 \mapsto \gamma\} \circ \{\delta_3 \mapsto \gamma\}$, still there is no simple non-identity substitution S' which satisfies $C \models_P S'(C)$. □

4.1.1 Constraint crowns and hardness of minimization

In Example 4.1.4 below we will re–introduce an important poset, called 2–*crown* (recall Section 2.4.1). This poset will play a major rôle both in the proof of Theorem 4.1.5 below and in the lower bound proof in Chapter 5, as well as in several examples later. Pratt and Tiuryn [62] studied so-called *n-crowns* to show that, for some finite posets P, the problem P-SAT is NP–complete (P-SAT is: given atomic constraint set C over P, determine if C is satisfiable in P.) This result holds for $P = n\text{-}crown$, for all $n \geq 2$, and it has been used by several researchers in the study of the complexity of subtype inference [71, 47, 9, 10]. Our use of crowns in this thesis is

[3] A related phenomenon is noticed by Pottier [59] for simplification with recursive types.

different, though, in that we will be interested in studying *constraint sets* that constitute 2–crowns when considered as ordered sets on the variables (in the obvious way: if, say, $\alpha \leq \beta \in C$, then α is below β in the ordering defined by C.) In the remainder of Part I of the thesis we will show several cases where such constraint sets are particularly complicated to simplify, much as crowns define hard satisfiability problems.

Example 4.1.4 (2-crowns)
Recall that the 2-crown is the poset with 4 elements $0, 1, 2, 3$ ordered as shown below (left):

Let C be the set of internal variables in a typing, $C = \{\alpha \leq \gamma, \beta \leq \gamma, \alpha \leq \delta, \beta \leq \delta\}$. If we regard C as a set ordered by inequalities, we see that it constitutes a 2-crown of variables (figure, right). The reader can verify that $C \mapsto^*_{\mathbf{g}} \{\alpha \leq \alpha\}$ by using the successive G-subsumptions $\beta \leq_{\mathbf{g}} \alpha, \gamma \leq_{\mathbf{g}} \alpha, \delta \leq_{\mathbf{g}} \alpha$. □

The example shows that crowns of internal variables are harmless. However, in sufficiently complex situations crowns of *observable* variables are not harmless, as our next observations show. Let G-minimization be the following problem: given an atomic constraint set C with a subset of $\mathsf{Var}(C)$ designated as observable, compute a g^+-normal form of C. Then it is not difficult to see that we have:

Theorem 4.1.5 *If $P \neq NP$, then G-minimization cannot be computed in polynomial time for any non–trivial partial order P.*

PROOF Fix the constraint set D shown in Figure 4.1 with variables α, β, γ, δ and η_1 through η_4, all designated as observable. Note that D can be regarded as three copies (in variables) of 2–crown which are "spliced together" at γ and δ, and at α and β; in particular, the middle part $\{\alpha, \beta, \gamma, \delta\}$ of D is isomorphic to 2–crown under the embedding $\pi = \{0 \mapsto \alpha, 1 \mapsto \gamma, 2 \mapsto \beta, 3 \mapsto \delta\}$.

Now let C be any atomic constraint set over 2–crown, and assume w.l.o.g. that $\mathsf{Var}(C) \cap \mathsf{Var}(D) = \emptyset$. We also assume w.l.o.g. that all constraint sets

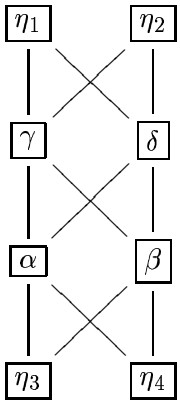


Figure 4.1: Constraint set D with all variables observable

considered are irreflexive (*i.e.*, there are no inequalities of the form $A \leq A$.) Translate C to $\overline{C}$ with

$$\overline{C} = D \cup \pi(C) \cup \{\alpha' \leq \eta_1, \alpha' \leq \eta_2, \eta_3 \leq \alpha', \eta_4 \leq \alpha' \mid \alpha' \in \mathsf{Var}(C)\}$$

with all $\alpha' \in \mathsf{Var}(C)$ designated as internal variables in $\overline{C}$. Let $\widetilde{C}$ be a G–minimal form of $\overline{C}$, where all reflexive inequalities $A \leq A$ have been removed. We claim that

$$(*) \qquad C \text{ is satisfiable in 2–crown if and only if } \widetilde{C} = D$$

Clearly, the theorem will follow from $(*)$ by NP–completeness of the satisfiability problem for 2–crown, since the condition $\widetilde{C} = D$ can be checked in polynomial time if G–minimal forms can be obtained in polynomial time.

To see that $(*)$ is true, suppose that C is satisfiable in 2–crown. Then $v(C)$ is true in 2–crown, for some valuation v. But then $\pi \circ v(C) = D$ (after removal of reflexive inequalities), because the middle part of D is isomorphic to 2–crown under π. This shows that there exists a G–minimized form of $\overline{C}$, where all internal variables are eliminated. It is easy to check that any substitution S eliminating all internal variables and satisfying $\overline{C} \models_P S(\overline{C})$ must map internal variables in $\overline{C}$ to variables in D (*i.e.*, unless P is trivial, there can be no constant $b \in P$ such that S maps any internal variable to b.) On the other hand, if S' is any other minimizing substitution, then $\overline{C} \models_P S'(\overline{C})$, hence $S(\overline{C}) \models_P S \circ S'(\overline{C})$, hence $\overline{C} \models_P S \circ S'(\overline{C})$, *i.e.*, $S \circ S'$ is

a sound substitution for G–minimizing $\overline{C}$. It follows that $\widetilde{C}$ has no internal variables left, by G–minimality of $\widetilde{C}$. But then $\widetilde{C} = D$ follows.

Conversely, suppose that $\widetilde{C} = D$. Then there is a substitution S such that S G-minimizes $\overline{C}$, with $S(\overline{C}) = \widetilde{C} = D$ (after removal of reflexive inequalities). Since all variables from C are forced in $\overline{C}$ to be below both of the observable variables η_1 and η_2 and above both of the observable variables η_3 and η_4, it follows from $\widetilde{C} = D$ that S must map all the variables in C to the middle part of D. But this shows that $\pi^{-1} \circ S$ satisfies C in 2–crown. □

Theorem 4.1.5 already holds for G–minimization defined in terms of $\vdash_P$, as shown in [65].

4.1.2 Formal lattice types

It is worth while to note (as is done in [59, 55]) that, if P happens to be a lattice L, then we can trivially eliminate all internal variables from typings, *provided* that we have *formal* meets or joins in the constraint language (we need only one or the other). Assuming that we can form constraints $A \vee A'$ (with the obvious semantics $v(A \vee A') = v(A) \vee_L v(A')$), then any set C can be transformed by the substitution S given by

$$S = \{\alpha \mapsto \bigvee \downarrow_C^{\mathsf{O}} (\alpha)\}_{\alpha \in \mathsf{Intv}(C)}$$

where $\mathsf{Obv}(C)$ and $\mathsf{Intv}(C)$ are the observable types and the internal variables, respectively, in C, relative to a given typing judgement with constraint set C, and where

$$\downarrow_C^{\mathsf{O}} (\alpha) = \{A \in \mathsf{Obv}(C) \mid C \models_P A \leq \alpha\}$$

The transformation is sound, because $C \models_L S(C)$, which, in turn, is true because $C \models_L \bigvee \downarrow_C^{\mathsf{O}} (\alpha) \leq \alpha$ and for $\alpha \leq \beta \in C$ with $\alpha, \beta \in \mathsf{Intv}(C)$, one has $\downarrow_C^{\mathsf{O}} (\alpha) \subseteq \downarrow_C^{\mathsf{O}} (\beta)$. So, for $\alpha \leq A \in C$ with $A \in \mathsf{Obv}(C)$, we have $C \models_L S(\alpha) \leq S(A)$; for $\alpha \leq \beta \in C$ with $\alpha, \beta \in \mathsf{Intv}(C)$, we have $C \models_L \bigvee \downarrow_C^{\mathsf{O}} (\alpha) \leq \bigvee \downarrow_C^{\mathsf{O}} (\beta)$; and for $A \leq \alpha \in C$ with $A \in \mathsf{Obv}(C)$, we have $A \in \downarrow_C^{\mathsf{O}} (\alpha)$, so $C \models_L S(A) \leq S(\alpha)$.

We will see later (Section 5.4.3) that there is an instance relation stronger than $\prec$ such that complete elimination of internal variables becomes possible without extending the constraint language. In contrast, Theorem 4.1.5

shows that, in the absence of formal lattice–operations, elimination of internal variables becomes highly non–trivial, and as we will see in Section 4.2, there are situations where not all internal variables of a typing can be eliminated under G–minimization. Theorem 4.1.5 could be taken as good evidence, therefore, that for subtyping over a lattice, one should allow (one of the) formal lattice operations in the language (or, alternatively, stronger instance relations than $\prec$; see Section 5.4.3).

4.2 S–simplification

S–simplification eliminates observable variables from constraint sets.

Definition 4.2.1 (S–subsumption, S–simplification) Given atomic constraint set C, type τ, variable α and atom A, we say that α is *S–subsumed by A in C and τ*, written $\alpha \sqsubseteq_{\mathsf{S}} A$, iff at least one of the following conditions is satisfied

1. *either*
   (a) $C \models_P A \leq \alpha$, and
   (b) α does not occur negatively in τ, and
   (c) $\Downarrow_C(\alpha) \setminus \{\alpha\} \subseteq \Downarrow_C(A)$
2. *or*
   (a) $C \models_P \alpha \leq A$, and
   (b) α does not occur positively in τ, and
   (c) $\Uparrow_C(\alpha) \setminus \{\alpha\} \subseteq \Uparrow_C(A)$

If $\mathbf{t} = C, \Gamma \vdash_P M : \tau$, with $\mathcal{D}(\Gamma) = \{x_1, \ldots, x_n\}$, then we define the type $\mathsf{Clos}_\Gamma(\mathbf{t})$ by

$$\mathsf{Clos}_\Gamma(\mathbf{t}) = \Gamma(x_1) \to \ldots \to \Gamma(x_n) \to \tau$$

We then define the reduction $\mapsto_{\mathsf{S}}$ on typings, called *S–simplification*: Let $\mathbf{t}_1 = C_1, \Gamma_1 \vdash_P M : \tau_1$ and $\mathbf{t}_2 = C_2, \Gamma_2 \vdash_P M : \tau_2$; then

$$\mathbf{t}_1 \mapsto_{\mathsf{S}} \mathbf{t}_2 \text{ iff } \begin{cases} (1) & \alpha \sqsubseteq_{\mathsf{S}} A \text{ in } C_1 \text{ and } \mathsf{Clos}_{\Gamma_1}(\mathbf{t}_1) \\ (2) & C_2 = C_1\{\alpha \mapsto A\} \\ (3) & \Gamma_2 = \Gamma_1\{\alpha \mapsto A\} \\ (4) & \tau_2 = \tau_1\{\alpha \mapsto A\} \\ (5) & \{\alpha \mapsto A\} \text{ is not a renaming on } \mathbf{t}_1 \end{cases}$$

We say that a typing is in *S–normal form*, if and only if there is no variable in its constraint set, which is S–subsumed by an atom. □

Notice that $\alpha \sqsubseteq_{\mathsf{S}} A$ implies $\alpha \sqsubseteq_{\mathsf{G}} A$ (but not vice versa). Hence, if $\mathbf{t}_1 \mapsto_{\mathsf{S}} \mathbf{t}_2$, we have $C_1 \models_P C_2$. Moreover, if α occurs only positively in $\mathsf{Clos}_{\Gamma_1}(\tau_1)$, then $C \models_P A \leq \alpha$ ensures that $C_1 \models_P \mathsf{Clos}_{\Gamma_1}(\tau_1) \leq \mathsf{Clos}_{\Gamma_2}(\tau_2)$; similarly for the case where α occurs only negatively. This shows that S–simplification is sound, *i.e.*, $\mathbf{t}_1 \mapsto_{\mathsf{S}} \mathbf{t}_2 \Rightarrow \mathbf{t}_1 \approx \mathbf{t}_2$.

The following lemma shows that constraint sets constituting 2–crowns of variables cannot be S–simplified, under certain conditions. We shall use this property in an essential way later, when we prove a lower bound on type size.

Lemma 4.2.2 *Let C be a set of the form*

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

constituting a 2–crown of variables. If $\mathbf{t} = C, \emptyset \vdash_P M : \tau$ is a typing in which α and β occur negatively in τ and γ and δ occur positively in τ, then $\mathbf{t}$ is in S–normal form, for any non–trivial poset P.

PROOF Notice first that, by the assumptions about positive and negative occurrences of the variables in C, for no valid S–subsumption $\alpha \sqsubseteq_{\mathsf{S}} A$ could A be a constant in P, provided that P is non–trivial. For instance, if we tried with $\alpha \sqsubseteq_{\mathsf{S}} b$, then (since α occurs negatively in τ), we would need to have $C \models_P \alpha \leq b$, which would evidently entail $\models_P \alpha \leq b$, so b would have to be top element in P; but then the transformed constraint set $C\{\alpha \mapsto \top\}$ would contain the inequality $\top \leq \gamma$, and the condition $C \models_P C\{\alpha \mapsto \top\}$ would not hold, unless P is the singleton set, showing $\alpha \sqsubseteq_{\mathsf{S}} b$ impossible for any non–trivial P. Similar arguments show that no other variable in C can be S–subsumed by a constant in P.

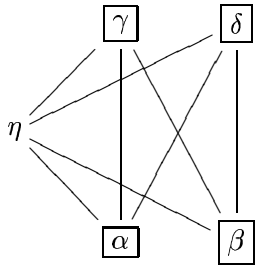
The lemma can now be verified by considering all possible S–subsumptions, which must be between variables in C. For instance, if one tries with $\alpha \sqsubseteq_{\mathsf{S}} \gamma$, then $\Uparrow_C(\alpha) \setminus \{\alpha\} \subseteq \Uparrow_C(\gamma)$ is forced by the definition of $\sqsubseteq_{\mathsf{S}}$; but this condition fails in any non–trivial poset P, which must contain two elements b_1, b_2 such that $b_1 \leq_P b_2$ and $b_2 \not\leq_P b_1$; clearly, we have $\delta \in \Uparrow_C(\alpha) \setminus \{\alpha\}$, but $\delta \notin \Uparrow_C(\gamma)$ since we can map α and δ to b_1 and γ to b_2 in some satisfying valuation. The other possible cases are symmetric to this one. □

Notice that the assumptions about negative occurrence of α, β and positive occurrence of γ, δ is necessary for the lemma. If, *e.g.*, all variables in C

occurred positively, then we could simplify the constraint set by mapping all variables to $\bot$ (assuming P has a bottom element.)[4]

The next example shows that there are indeed situations where we cannot eliminate all internal variables, even under the combination of G–minimization and S–simplification.

Example 4.2.3 Let P be any non–trivial poset (*i.e.*, P not discretely ordered.) Then there are two distinct elements $0, 1 \in P$ with $0 < 1$. Let C be the constraint set over P:



We assume that C occurs in a typing $\mathbf{t}$ where the variables α, β, γ, δ are observable with α and β having negative occurrences and γ and δ having positive occurrences. The variable η is internal.

The set C is G–simplified (hence also G–minimized, since there is just one internal variable) over any non–trivial order P. This shows that, indeed, there are situations where not all internal variables can be eliminated. We leave it to the reader to check that there exists no atom A such that $\eta \sqsubseteq_{\mathsf{g}} A$ with respect to C (for $A \in P$ we must use that P is not a singleton set). The set C cannot be S–simplified either, by Lemma 4.2.2. It is not difficult to check, then, that the constraint set C cannot be simplified at all, *i.e.*, there is no non–renaming substitution S such that $(\mathbf{t} \prec_S \mathbf{t}$. □

The reader has now seen a number of examples involving 2–crowns defined by constraint sets, and one may wonder whether such sets can actually occur in real typings of terms (programs). The fact is that they can; in Chapter 5 we will construct terms that generate large numbers of crowns.

[4] This is one place where a system built on $\models_P$ differs from a system built on $\vdash_P$; in [65], where we used the relation $\vdash_P$, we did not have to take into account the polarities of variables, since no crown of observable variables can be S–simplified in any non–trivial order P, no matter how the variables occur in the type.

4.2.1 Partial completeness of S–simplification

We now prove the main technical result of this chapter. It shows that S–simplification has an interesting partial completeness property. We show that, for typings of a certain form, S–simplification leads to minimal typings in the sense of Definition 3.2.7. The essential property needed to prove this is contained in the following proposition.

Proposition 4.2.4 *Let* $\mathbf{t} = C, \Gamma \vdash_P M : \tau$ *and* $\tau' = \mathit{Clos}_\Gamma(\mathbf{t})$. *Assume* S *is not the identity on* $\mathit{Var}(\mathbf{t})$ *with*

(*i*) $C \models_P S(C)$

(*ii*) $C \models_P S(\tau') \leq \tau'$

(*iii*) $\mathit{Supp}(S) \subseteq \mathit{Obv}(t)$

(*iv*) C *is acyclic*

Then there exists $\alpha \in \mathit{Supp}(S)$ *such that* $\alpha \leq_S S(\alpha)$ *with respect to* C *and* τ'.

PROOF The proof is by contradiction, so suppose, under the assumptions of the proposition, that

$$\neg\exists\alpha \in \mathsf{Supp}(S).\ \alpha \leq_{\mathsf{S}} S(\alpha) \text{ wrt. } C \text{ and } \tau' \tag{4.1}$$

Pick any $\alpha \in \mathsf{Supp}(S)$ (by assumption $\mathsf{Supp}(S) \neq \emptyset$), so we have $\alpha \neq S(\alpha)$. By (*iii*), α must occur in τ'. It is then easy to verify, by induction in τ' that (*ii*) implies that

1. if α occurs negatively in τ', then $C \models_P \alpha \leq S(\alpha)$, and
2. if α occurs positively in τ', then $C \models_P S(\alpha) \leq \alpha$

Now, since C is acyclic, these two statements cannot both be true, since if they were, then we would have $C \models_P \alpha = S(\alpha)$, implying that C is cyclic (remember that we have $\alpha \neq S(\alpha)$). On the other hand, since α occurs in τ', it has either negative or positive occurrences. Therefore we can conclude that *either* $C \models_P S(\alpha) \leq \alpha$ with α not occurring negatively in τ', *or* else $C \models_P \alpha \leq S(\alpha)$ with α not occurring positively in τ'. Assume that we have (the alternative case is similar)

$$(*) \qquad \begin{array}{l} C \models_P S(\alpha) \leq \alpha \\ \text{with } \alpha \text{ not occurring negatively in } \tau' \end{array}$$

By $C \models_P S(\alpha) \leq \alpha$ and $S(\alpha) \neq \alpha$ we get that, if $S(\alpha)$ is not a constant, then $S(\alpha)$ must be a variable occurring in C, by Lemma 3.4.3. Therefore $\{\alpha \mapsto S(\alpha)\}$ cannot be a renaming on $\mathbf{t}$. By (4.1) we must then have $\Downarrow_C(\alpha) \setminus \{\alpha\} \not\subseteq \Downarrow_C(S(\alpha))$, since otherwise we should have $\alpha \sqsubseteq_{\mathsf{S}} S(\alpha)$. So there must be an atomic type $A_1 \in \Downarrow_C(\alpha) \setminus \{\alpha\}$ such that

$$A_1 \notin \Downarrow_C(S(\alpha)) \tag{4.2}$$

In particular, we therefore have

$$A_1 \neq S(\alpha), A_1 \neq \alpha \text{ and } C \models_P A_1 \leq \alpha \tag{4.3}$$

By the Substitution Lemma, we then have $S(C) \models_P S(A_1) \leq S(\alpha)$, and so, by (i), we also have

$$C \models_P S(A_1) \leq S(\alpha) \tag{4.4}$$

If $A_1 = S(A_1)$, then by (4.4) it would follow that $C \models_P A_1 \leq S(\alpha)$, hence $A_1 \in \Downarrow_C(S(\alpha))$ in contradiction with (4.2). Therefore we must have

$$A_1 \neq S(A_1) \tag{4.5}$$

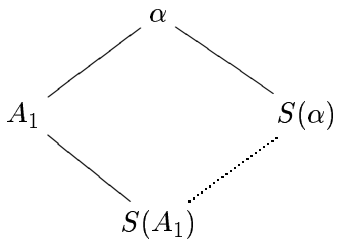
from which it follows that

$$A_1 \text{ is a variable in } \mathsf{Supp}(S) \tag{4.6}$$

Now, by (4.6) and assumption (iii), A_1 is an observable variable, occurring in τ'. Then (ii) implies that A_1 and $S(A_1)$ must be comparable under the hypotheses C. We already know (4.5) that $A_1 \neq S(A_1)$, and if $C \models_P A_1 \leq S(A_1)$, then it would follow by (4.4) that $C \models_P A_1 \leq S(\alpha)$ in contradiction with (4.2). We must therefore conclude that

$$\begin{array}{l} C \models_P S(A_1) \leq A_1 \\ \text{with no negative occurrence of } A_1 \text{ in } \tau' \end{array} \tag{4.7}$$

Summing up so far, we now have the situation



where the dotted line means "less than or equal to" and the solid lines mean "strictly less than", and the relations shown hold under entailment with the hypotheses C. But this, together with (4.7) shows that the situation described in $(*)$ now holds with A_1 and $S(A_1)$ in place of α and $S(\alpha)$, and all the reasoning starting from $(*)$ can therefore be repeated for A_1 and $S(A_1)$, leading, again, to the existence of an atom A_2 (which must be a variable in C) with $C \models_P A_2 \leq A_1$ and $A_2 \neq A_1$ etc. Hence, by repetition of the argument starting at $(*)$, we obtain an arbitrarily long strictly decreasing chain (under entailment with C) of variables in C:

$$\alpha > A_1 > A_2 > \ldots$$

implying that C is cyclic and thereby contradicting (iv). We have now reached the desired contradiction, and we must therefore reject the assumption (4.1), and the proof of the Proposition is complete. □

Using Proposition 4.2.4, we can now prove the following result.

Theorem 4.2.5 *(Completeness of S wrt. observables)*
If $\mathbf{t} = C, \Gamma \vdash_P M : \tau$ *is an S–simplified typing with* C *acyclic and* $\mathsf{Var}(C) \subseteq \mathsf{Obv}(\mathbf{t})$*, then the typing* $\mathbf{t}_0 = \mathsf{Red}(C), \Gamma \vdash_P M : \tau$ *is minimal.*

PROOF We will show that any typing $\mathbf{t}$ satisfying the conditions of the theorem must be fully substituted. Now suppose that S is a non–renaming substitution on $\mathbf{t}$ such that $S(\mathbf{t}) \prec \mathbf{t}$. We can assume w.l.o.g. that S is the identity on variables not appearing in $\mathbf{t}$. Since $\mathsf{Var}(C) \subseteq \mathsf{Obv}(\mathbf{t})$ it follows that $\mathsf{Supp}(S) \subseteq \mathsf{Obv}(\mathbf{t})$. Since $S(\mathbf{t}) \prec \mathbf{t}$, there is a substitution S' such that

1. $C \models_P S' \circ S(C)$
2. $C \models_P S' \circ S(\mathsf{Clos}_\Gamma(\tau)) \leq \mathsf{Clos}_\Gamma(\tau)$

Since S is not a renaming on $\mathbf{t}$, it follows that $S' \circ S$ is not a renaming on $\mathbf{t}$, and then Proposition 4.2.4 shows that $\mathbf{t}$ is not S–simplified. This contradicts the assumption that $\mathbf{t}$ is S–simplified. Therefore, any substitution S such that $S(\mathbf{t}) \in [\mathbf{t}]$ must be a renaming on $\mathbf{t}$. This establishes that $\mathbf{t}$ is fully substituted. Theorem 3.5.3 then implies that $\mathbf{t}_0$ is minimal in its equivalence class. □

This theorem allows us to conclude that certain typings are minimal. Since S–simplification requires some fairly strong conditions to be satisfied by the

typing, it may be an easy matter to establish that the conditions of the theorem are satisfied. Theorem 4.2.5 will be used in Chapter 5, where we prove an exponential lower bound for the worst case size of constraint sets and types in principal typings of $\lambda_{\leq}(\models_P)$.

Chapter 5

The size of principal typings

This chapter investigates the asymptotic worst case dag–size of principal typings in atomic subtyping systems. We prove a tight worst case exponential lower bound for the dag-size of both constraint sets and types of principal typings in *atomic* subtyping systems, relative to any specific instance relation proposed so far in the literature. To the best of our knowledge, the only lower bound result previously proven for type–size in subtyping systems is the linear lower bound for a whole class of so–called sound instance relations shown in [37] (see Section 1.5). Moreover, we separate the known instance relations by studying the relative power, with respect to simplifications, of a series of increasingly large instance relations. We show that each larger relation validates simplifications which, in the extreme, can yield an exponential compression in the size of principal typings, in comparison with the smaller relations. We argue that the exponential lower bound is also significant for non–atomic systems. The proof of the exponential lower bound as well as the separation results draw on the results of Chapter 3 and Chapter 4 with Theorem 4.2.5.

For the simple typed λ–calculus, λ_s, we know (see [42]) that while *textual* type size can be exponential, *dag-size* (see, *e.g.*, [42] and [54, Chapter 11.3]) is at most linear. The basic property responsible for this is that the type system of λ_s imposes enough equality constraints, in the sense of [74], so that sharing yields exponential succinctness. In the case of ML, succinctness of `let`-expressions leads to doubly exponential *textual* size of types and exponential *dag*-size, in the worst case (see [42, 54] with further references.) Simple subtyping ($\lambda_{\leq}$ for short) as studied here, is a system based on *in*equalities, and, as shown below, this *alone* leads to exponential

dag-size of typings (constraint sets as well as of types), for the reason that exponentially many distinct variables must be present in a principal typing, in the worst case. The situation is summarised in the table:

	λ_s	*ML*	$\lambda_{\leq}$
text	2^n	2^{2^n}	$\mathbf{2^n}$
dag	n	2^n	$\mathbf{2^n}$

The table shows that the distinctive property of the subtyping system is the absence of dag–compression. Both λ_s and ML have exponentially succinct dag–types, even though the ML system lies an exponential higher than simple types. Needless to say, the system $\lambda_{\leq}$ has no quantified types and no `let`–construct, so the loss of dag–compression, in passing from λ_s to $\lambda_{\leq}$, must be explained solely in terms of the presence of subtyping. This phenomenon is a reflection of the intuitive property that subtyping introduces a higher "degree of freedom" in type structure. However, up until now, no "hard" evidence has ever been given to support this intuition.

This result is interesting for at least the following two reasons.

1. It gives a complexity theoretic foundation to the observed phenomenon that subtyping systems are hard to handle in practice.

2. It identifies an intrinsic limit to how much type-simplification techniques can possibly achieve for atomic subtyping, relative to any notion of instance suggested so far, no matter how clever we are at inventing simplifications validated by any of those instance relations.

The remainder of this chapter is organized as follows. Section 5.1 through Section 5.3 prove the exponential lower bound for the system $\lambda_{\leq}(\models_P)$, which uses the instance relation $\prec$. The idea here is not just to prove the lower bound, but to do it in such a way that we can separate the power of $\prec$ from the power of other notions of instance. The proof must therefore be "tailored" for the system $\lambda_{\leq}(\models_P)$. Section 5.4 studies a more powerful simplification framework called "semantic subtyping", using the to our knowledge

strongest known notion of instance, called $\prec_{sem}$. In Section 5.4 we prove an exponential lower bound for the size of principal typings relative to $\prec_{sem}$, and we separate the relations $\prec_{syn}$, $\prec$ and $\prec_{sem}$ with respect to simplification, showing that, in extreme cases, the more powerful relation may yield exponentially more succinct principal typings than the less powerful ones. Section 5.5 shows that non–linearity is an essential property of terms with large principal typings. The remaining sections are less technical, focusing on the significance of the lower bound results. Section 5.6 discusses the difference between the typability problem and the problem of representing principal typings. Section 5.7 argues that the lower bound results on type size remains significant for non–atomic systems.

5.1 Preliminaries

We begin now to prove the exponential lower bound for $\lambda_{\leq}(\models_P)$. The proof has two core parts. One is the construction of a series of terms $\mathbf{Q}_n$, with the intention that, for all $n \geq 0$, the principal typing of $\mathbf{Q}_n$ generated by a certain standard procedure has the form $C_n, \emptyset \vdash_P \mathbf{Q}_n : \tau_n$, where C_n contains more than 2^n distinct type variables and τ_n contains more than 2^n distinct type variables. The second main ingredient in the proof is Theorem 4.2.5 and the characterization of minimal typings of Section 3.2, which are employed in order to prove that the same property in fact holds for *all* principal typings of $\mathbf{Q}_n$, viz. that *any* principal typing must have a number of variables in both constraint set and type which is exponentially dependent on the size of the terms.

For the purpose of the following development we shall first assume that we have a conditional construct with the typing rule

$$[\mathit{if}] \quad \frac{C, \Gamma \vdash_P M : \mathtt{bool} \quad C, \Gamma \vdash_P N : \tau \quad C, \Gamma \vdash_P Q : \tau}{C, \Gamma \vdash_P \mathit{if}\ M\ \mathit{then}\ N\ \mathit{else}\ Q : \tau}$$

Moreover, we shall assume pairs $\langle M, N \rangle$ and product types $\tau \times \tau'$ with the usual typing rule

$$[\mathit{pair}] \quad \frac{C, \Gamma \vdash_P M_1 : \tau_1 \quad C, \Gamma \vdash_P M_2 : \tau_2}{C, \Gamma \vdash_P \langle M_1, M_2 \rangle : \tau_1 \times \tau_2}$$

together with projections π_1 and π_2 using the standard typing rules

$$[\mathit{proj1}] \quad \frac{C, \Gamma \vdash_P M : \tau_1 \times \tau_2}{C, \Gamma \vdash_P (\pi_1\ M) : \tau_1} \qquad [\mathit{proj2}] \quad \frac{C, \Gamma \vdash_P M : \tau_1 \times \tau_2}{C, \Gamma \vdash_P (\pi_2\ M) : \tau_2}$$

The subtype ordering is lifted co-variantly to pairs in the usual way, such that $\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'$ holds if and only if $\tau_1 \leq \tau_1'$ and $\tau_2 \leq \tau_2'$ both hold. The additional constructs are introduced only because it is easier to understand the essence of the lower bound proof, and we show how to eliminate them form the proof.

5.1.1 Standard procedure

We will be analyzing the form of principal typings of certain terms. We use the fact, shown in [53], that a principal typing can always be obtained by the following *standard procedure*:

(1) first extract subtyping constraints only at the *leaves* of the term, *i.e.*, coercions (applications of the rule [*sub*], see below) are applied to variables and constants only, and then

(2) perform a *match-step* in which a most general matching substitution (see [53] for details) is applied to the extracted constraint set (the match-step may fail, but if so then the term has no typing with atomic subtyping at all) and finally

(3) *decompose* the matching constraints into atomic constraints, using the Decomposition Lemma (any matching set can be decomposed)

Once steps (1) through (3) have been performed, we can then apply transformations such as G and S to the typing, without losing principality.

5.1.2 Coercions and completions

We shall sometimes indicate the form of a typing derivation by *completions*. Completions are terms with explicit subtyping *coercions* and type assumptions for bound variables. Coercions, as used here, serve the purely logical purpose of indicating what a typing proof looks like. A coercion from τ to τ' applied to a term M will be written as $\uparrow_{\tau}^{\tau'} M$, so, for instance, the completion

$$\lambda x : \alpha.\, \uparrow_{\alpha}^{\beta} x$$

encodes the typing judgement $\{\alpha \leq \beta\}, \emptyset \vdash_P \lambda x.x : \alpha \to \beta$ as well as a derivation of that judgement; coercions indicate where the subsumption rule [*sub*] is used in the term and to which types it is applied.

5.2 The construction

In order to prove the lower bound on the size of typings, we want to construct a series of terms $\mathbf{Q}_n$, such that for all $n \geq 0$, any principal typing of $\mathbf{Q}_n$ must contain an exponential number of distinct type variables in both constraint set and type. It is not difficult to write down a series of lambda terms that will produce types of textual size exponential in the size of the terms. A well known (see [42, 54]) series of terms with this property is

$$\mathbf{D}^n M$$

the n-fold application to M of the "duplicator"

$$\mathbf{D} = \lambda z.\langle z, z\rangle$$

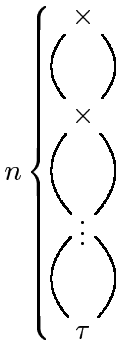
Sure enough, if we extract constraints from these terms in a standard, mechanical way, then we may get an exponentially large number of subtyping constraints. But that is only one of infinitely many possible representations of the principal typing. And in fact, the principal typings of these terms, as derived by the standard procedure in the subtype system, can be simplified (using, in fact, only G-and S-simplification) in such a way that we get back their simple types of linear dag-size, as the reader may like to verify. Assuming that we have

$$\emptyset, \emptyset \vdash_P M : \tau$$

Then

$$\emptyset, \emptyset \vdash_P \mathbf{D}^n M : \underbrace{\tau \times \tau \times \ldots \times \tau}_{2^n}$$

so that dag–representation of the type yields exponential succinctness:



The situation here is no different from what we have in simple types, and so the terms $\mathbf{D}^n$ will not give us what we are after.

This leads to the idea that, in order to get an exponential blow-up in dag-size of types in *every possible* principal typing, we must somehow produce a series of terms with the following properties

1. The terms must in some uniform way generate exponentially large minimal typings, and
2. The terms must be such that we can easily tell what their minimal typings look like

To achieve these goals, we construct the terms $\mathbf{Q}_n$ essentially in such a way that the standard procedure will generate an exponential number of 2-*crowns* of *observable* variables in the constraint sets. We know from Lemma 4.2.2 that observable 2-*crowns* cannot be S–simplified, if the variables at the bottom of the crowns occur negatively and those at the top occur positively in the type of the judgement. We shall construct our terms $\mathbf{Q}_n$ in such a way that this becomes true of the crowns in the constraint sets. We can then invoke Theorem 4.2.5 to argue that the *minimal* typing of the $\mathbf{Q}_n$ must have exponentially many distinct variables. From this the lower bound will follow from Proposition 3.6.1.

The terms $\mathbf{Q}_n$

We proceed to explain the construction of the terms $\mathbf{Q}_n$. Let $cond_{x,y}$ denote the expression with two free variables x and y, given by

$$cond_{x,y} = \mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle x, y\rangle\ \mathit{else}\ \langle y, x\rangle$$

For a term variable f, define the expression $\mathbf{P}_f$ with free variable f as follows:

$$\begin{array}{rcl} \mathbf{P}_f & = & \lambda z.\mathbf{K} \\ & & (\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle z, \langle \pi_2\ z, \pi_1\ z\rangle\rangle \\ & & \mathit{else}\ \langle\langle \pi_2\ z, \pi_1\ z\rangle, z\rangle) \\ & & (f\ z) \end{array}$$

where $\mathbf{K}$ is the combinator $\lambda x.\lambda y.x$. Let $f_1, f_2, \ldots$ be an enumeration of infinitely many distinct term variables, and let N be any expression; then define, for $n \geq 0$, the expression $\mathbf{P}^n N$ recursively by setting

$$\begin{array}{rcl} \mathbf{P}^0 N & = & N, \\ \mathbf{P}^{n+1} N & = & \mathbf{P}_{f_{n+1}}(\mathbf{P}^n N) \end{array}$$

Write $\lambda f_{[n]}.M = \lambda f_n.\ldots.\lambda f_1.M$ for $n \geq 1$ and $\lambda f_{[0]}.M = M$. Now we can define the series of terms $\mathbf{Q}_n$ for $n \geq 0$ by setting

$$\mathbf{Q}_n = \lambda f_{[n]}.\lambda x.\lambda y.\mathbf{P}^n cond_{x,y}$$

So, for instance, we have

$$\mathbf{Q}_0 = \lambda x.\lambda y.cond_{x,y}$$

and

$$\begin{array}{rcl} \mathbf{Q}_1 & = & \lambda f_1.\lambda x.\lambda y.(\lambda z.\mathbf{K} \\ & & (\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle z, \langle \pi_2\ z, \pi_1\ z\rangle\rangle \\ & & \mathit{else}\ \langle\langle \pi_2\ z, \pi_1\ z\rangle, z\rangle) \\ & & (f_1 z)) \\ & & cond_{x,y} \end{array}$$

The behaviour of $\mathbf{Q}_n$

Before delving into the details of the proof that the terms $\mathbf{Q}_n$ behave as claimed, let us give an intuitive explanation of what is going on.

Consider the term $\mathbf{Q}_0$. It is easy to see that a principal typing of $\mathbf{Q}_0$ has the form

$$\mathbf{t}_0 = C_0, \emptyset \vdash_P \lambda x.\lambda y.cond_{x,y} : \alpha \to (\beta \to \gamma_1 \times \gamma_2)$$

with

$$C_0 = \{\alpha \leq \gamma_1, \alpha \leq \gamma_2, \beta \leq \gamma_1, \beta \leq \gamma_2\}$$

This typing is derived by the *standard procedure* described earlier, where G-simplification has been performed to eliminate internal variables. The resulting typing derivation is given by the completion:

$$\begin{array}{l} \lambda x : \alpha.\lambda y : \beta.\mathit{if}\ \mathtt{true} \\ \mathit{then}\ \langle \uparrow_\alpha^{\gamma_1} x, \uparrow_\beta^{\gamma_2} y\rangle \\ \mathit{else}\ \langle \uparrow_\beta^{\gamma_1} y, \uparrow_\alpha^{\gamma_2} x\rangle \end{array}$$

All variables in C_0 are observable, and C_0 is the 2-crown shown in Figure 5.1. The set C_0 is evidently acyclic. Moreover, the variables at the bottom of the crown (*i.e.*, the variables α and β) occur negatively in the type of $\mathbf{t}_0$, and the variables at the top of the crown (*i.e.*, the variables γ_1 and γ_2) occur positively in the type of $\mathbf{t}_0$. It follows from Lemma 4.2.2 that $\mathbf{t}_0$ is S–simplified; Theorem 4.2.5 then allows us to conclude that $\mathbf{t}_0$ is a minimal judgement for $\mathbf{Q}_0$. Notice that the mechanism responsible for the generation

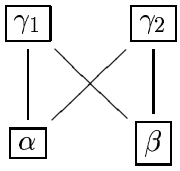


Figure 5.1: Constraint crown for $n = 0$

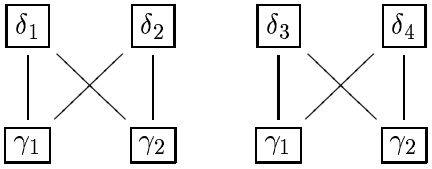


Figure 5.2: Constraint crowns generated by $\mathbf{P}_{f_1}$

of the crown is the *alignment*, within the conditional, of permutations of the same pair. Two objects are "aligned" if they appear in corresponding positions in a pair, one object in each branch of the same conditional. Such two objects are forced to have a common supertype, by the typing rule for the conditional. Thus, this rule forces a common supertype $\gamma_1 \times \gamma_2$ for both of the permuted pairs, and thereby it forces the "crown–inequalities" $\alpha \times \beta \leq \gamma_1 \times \gamma_2$ and $\beta \times \alpha \leq \gamma_1 \times \gamma_2$; these inequalities generate the crown, when they are decomposed.

To understand what happens at $n = 1$, consider the term $\mathbf{P}_{f_1}$ defined by

$$\begin{array}{rcl} \mathbf{P}_{f_1} & = & \lambda z.\mathbf{K} \\ & & (\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle z, \langle \pi_2\ z, \pi_1\ z\rangle\rangle \\ & & \mathit{else}\ \langle\langle \pi_2\ z, \pi_1\ z\rangle, z\rangle) \\ & & (f_1\ z) \end{array}$$

Assuming that $\mathbf{P}_{f_1}$ is applied to an object of type $\gamma_1 \times \gamma_2$, it is easy to verify that the following completion represents a principal typing of $\mathbf{P}_{f_1}$:

$$\begin{array}{rcl} \mathbf{P}_{f_1} & = & \lambda z : \gamma_1 \times \gamma_2.\mathbf{K} \\ & & (\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle \uparrow^{\delta_1\times\delta_2}_{\gamma_1\times\gamma_2} z, \langle \pi_2\ \uparrow^{\gamma_1\times\delta_3}_{\gamma_1\times\gamma_2} z, \pi_1\ \uparrow^{\delta_4\times\gamma_2}_{\gamma_1\times\gamma_2} x\rangle\rangle \\ & & \mathit{else}\ \langle\langle \pi_2\ \uparrow^{\gamma_1\times\delta_1}_{\gamma_1\times\gamma_2} z, \pi_1\ \uparrow^{\delta_2\times\gamma_2}_{\gamma_1\times\gamma_2} z\rangle, \uparrow^{\delta_3\times\delta_4}_{\gamma_1\times\gamma_2} x\rangle) \\ & & (f_1\ z) \end{array}$$

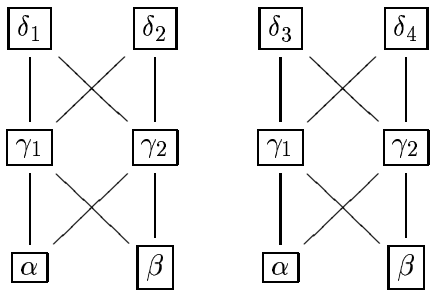


Figure 5.3: Constraint crowns for $n = 1$

at the type

$$\gamma_1 \times \gamma_2 \to (\delta_1 \times \delta_2) \times (\delta_3 \times \delta_4)$$

This typing appears after G–minimizing the typing generated by the standard procedure. The coercion set corresponding to the coercions shown in the term constitute the double 2–crown shown in Figure 5.2. It is not too difficult to verify that, combining the completion shown for $cond_{x,y}$ with the one shown for $\mathbf{P}_{f_1}$, results in a completion of $\mathbf{Q}_1$ with constraint set C_1 shown in Figure 5.3, at the type

$$((\gamma_1 \times \gamma_2) \to \eta) \to \alpha \to \beta \to ((\delta_1 \times \delta_2) \times (\delta_3 \times \delta_4))$$

where $(\gamma_1 \times \gamma_2) \to \eta$ is the type assumed for the variable f_1 in $\mathbf{Q}_1$. We notice that we now have two "towers" of 2–crowns, with new variables (δ_1 thru δ_4) sitting at the top of two copies of the crown generated for $n = 0$. It still holds that α and β sit at the bottom of each crown, and α and β occur negatively in the type of $\mathbf{Q}_1$, while the other variables occur positively; and in particular, all variables in C_1 are observable.

The general pattern should now be discernible at $n = 2$, referring to Figure 5.4. The term $\mathbf{Q}_2$ will force that the previous crowns get doubled, resulting in $2^n(= 4)$ "towers" of 2–crowns and introducing $2^{n+1}(= 8)$ new variables (named v_1 thru v_8 in Figure 5.4) sitting at the top of the previous towers of crowns generated at $n = 1$, resulting in a total of $2^{n+2}(= 16)$ distinct variables, organized in 2^n "towers" of crowns, as shown in Figure 5.4. It remains invariant that α and β sit at the bottom of all the crowns, occurring negatively in the type, while the remaining variables occur positively, and therefore the crowns cannot be eliminated from the principal typing

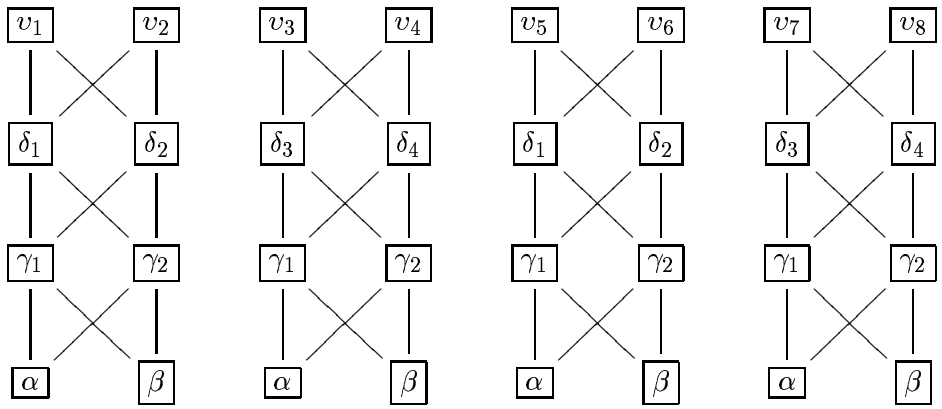


Figure 5.4: Constraint crowns for $n = 2$

(*i.e.*, there is a minimal principal typing in which the crowns are present.) The construction of $\mathbf{Q}_n$ uses the application $(f_n\ z)$ in order to force that intermediate variables in the crowns become observable; this is in fact not strictly necessary, but it simplifies the correctness proof for the construction. The fact that the towers of crowns grow "incrementally", by continuing, at step $n+1$, to build new crowns on top of the towers of crowns generated at step n, makes it possible to organize an induction proof (induction in n) of the correctness of this construction.

5.3 Exponential lower bound proof

The main lemma in the proof of the lower bound is the following, which formalizes the intuitive explanation given above.

Lemma 5.3.1 (Main Lemma) *Let P be any non–trivial poset. For all $n \geq 0$, there is a minimal principal typing for $\mathbf{Q}_n$ having the form $C_n, \emptyset \vdash_P \mathbf{Q}_n : \tau^{[n]}$, where all variables in C_n are observable, and C_n contains at least 2^{n+1} distinct variables.*

PROOF See Appendix A.3 □

In Appendix A.3 we show that the lower bound construction can be made in pure λ–calculus, without conditional, pairing and projection. We can now prove

Theorem 5.3.2 *For any non–trivial poset P of base types it holds for arbitrarily large n, that there exist closed terms of length $O(n)$ such that any principal typing (wrt. $\prec$) for the terms has a constraint set containing $2^{\Omega(n)}$ distinct observable type variables and a type containing $2^{\Omega(n)}$ distinct type variables.*

PROOF Take the series of terms $\mathbf{Q}_n$; clearly, the size of $\mathbf{Q}_n$ is of the order of n, and Lemma 5.3.1 shows that $\mathbf{Q}_n$ has a minimal principal typing $\mathbf{t}$ with constraint set containing more than 2^n distinct variables and a type containing more than 2^n distinct variables. Any other principal typing $\mathbf{t}'$ satisfies $\mathbf{t} \approx \mathbf{t}'$, and hence $\mathbf{t} \ll \mathbf{t}'$, by minimality of $\mathbf{t}$. By Proposition 3.6.1, this entails that $\mathbf{t}'$ must have at least as many distinct type variables in both constraint set and type as $\mathbf{t}$. □

The theorem immediately implies that the *dag-size* of constraint sets as well as of types in principal typings are of the order $2^{\Omega(n)}$ in the worst case. Since it follows from standard type inference algorithms such as those of [53] and [28] that a principal atomic typing can be obtained by extracting a set C of (possibly non–atomic) constraints of size linear in the size of the term followed by a match–step which expands and decomposes C to atomic constraints under at most an exponential blow-up, we have in fact a tight exponential bound:

Corollary 5.3.3 *For any non–trivial poset P of base types, the dag-size of constraint sets as well as of types in atomic principal typings (wrt. $\prec$) is of the order $2^{\Theta(n)}$ in the worst case.*

There are two limitations on the lower bound result proven in this section. The first limitation is that we were pre–supposing a particular notion of instance, $\prec$. The second limitation is that the type system is assumed to be atomic. While the lower bound proof obviously depends on both of these assumptions, we will argue below (Section 5.4 and Section 5.7) that the result remains significant for systems that do not obey these restrictions.

5.4 A hierarchy of instance relations

It is possible to envisage instance relations stronger (that is, larger) than $\prec$, and it is a theoretical possibility that there might exist meaningful instance relations strong enough to validate simplifications of such power that the exponential lower bound could be by–passed. The rationale here is that

a larger instance relation allows more typings to become equivalent, hence there will be more ways of presenting principal typings, and hence there might exist more (possibly exponentially) *succinct* presentations. However, as we will argue in this section, it is doubtful that the exponential lower bound can be overcome this way.

5.4.1 Instance relations

In this section we present a catalogue of increasingly large instance relations. The catalogue covers all instance relations suggested so far in the literature. In the sequel, we will work with the notational convention that, by default, $\mathbf{t}_i$ denotes a typing judgement of the form $C_i, \Gamma_i \vdash_P M : \tau_i$, and $\mathbf{t}'$ denotes a judgement of the form $C', \Gamma' \vdash_P M : \tau'$.

Weak instance relation ($\prec_{weak}$)

For completeness, we mention the weakest (smallest) known instance relation, introduced by Mitchell [53]; it was defined in Section 1.5, where it was called $\prec_{weak}$. As mentioned there, it does not validate even very trivial simplifications, and we will not consider it further.

Syntactic instance relation ($\prec_{syn}$)

A stronger (larger) notion of instance is the following "syntactic" version of $\prec$, called "lazy instance" by Fuh and Mishra [28], who introduced it precisely for the sake of validating more powerful simplifications. It arises from the definition of $\prec$ by exchanging the relation $\models_P$ with the relation $\vdash_P$. We denote the syntactic relation $\prec_{syn}$, and we have $\mathbf{t} \prec_{syn} \mathbf{t}'$ if and only if there exists a substitution S such that

1. $C' \vdash_P S(C)$
2. $C' \vdash_P S(\tau) \leq \tau'$
3. $\forall x \in \mathcal{D}(\Gamma).\ C' \vdash_P \Gamma'(x) \leq S(\Gamma(x))$

Clearly, $\prec_{weak} \subseteq \prec_{syn}$.

Model theoretic instance relation ($\prec$)

The relation $\prec$ defined in Definition 3.1.1 and studied in previous chapters is an intermediate relation, obviously larger (relating more typings) than

$\prec_{syn}$. We call it "model theoretic" instance here, to distinguish it from the others.

Semantic instance relation ($\prec_{sem}$)

In [73], Trifonov and Smith introduced and studied a highly natural and very powerful subsumption relation, called "semantic subtyping", on polymorphic qualified type schemes of the form $\forall\vec{\alpha}.\Gamma \Rightarrow \tau/C$. This notion of subsumption is closely related to the second order polymorphic subsumption relation introduced by Mitchell [52]. Trifonov and Smith considered subsumption in the model of infinite trees equipped with the Amadio–Cardelli order [5]. However, the relation makes sense for other models as well, and we consider here a version appropriate for the present setting of simple subtypes over a partial order of ground types.

A combination of a simple type τ and a constraint set C, written τ/C, will be called a *qualified type*. The set of *ground instances* of a qualified type τ/C is the set

$$\{v(\tau) \mid v \models_P C\}$$

that is, an instance is produced by applying a valuation satisfying C to τ. Relative to a poset P of type constants we order qualified types by a subsumption relation called $\prec_{sem}$, by setting $\tau/C \prec_{sem} \tau'/C'$ if and only if, for every instance of τ'/C' there is a smaller instance of τ/C. That is, we have $\tau/C \prec_{sem} \tau'/C'$ if and only if

$$\forall v' \models_P C'.\ \exists v \models_P C.\ \ v(\tau) \leq v'(\tau')$$

We can generalize the order $\prec_{sem}$ to arbitrary typings, by setting $\mathbf{t} \prec_{sem} \mathbf{t}'$ if and only if

$$\forall v' \models_P C.\ \exists v \models_P C.\ v'(\Gamma') \leq v(\Gamma) \wedge v(\tau) \leq v'(\tau')$$

where we order typing contexts by setting $\Gamma' \leq \Gamma$ if and only if $\mathcal{D}(\Gamma) \subseteq \mathcal{D}(\Gamma')$ and $\Gamma'(x) \leq \Gamma(x)$ for all $x \in \mathcal{D}(\Gamma)$.

The relation $\prec$ can be regarded as a relation between qualified types in the obvious way. We clearly have $\prec \subseteq \prec_{sem}$, because $\tau/C \prec \tau'/C'$ evidently holds if and only if

$$\exists S.\ \forall v \models_P C'.\ v \circ S \models_P C \wedge v(\Gamma') \leq v \circ S(\Gamma) \wedge v \circ S(\tau) \leq v(\tau')$$

Summing up, we have the increasing series of relations:

$$\prec_{weak} \subseteq \prec_{syn} \subseteq \prec \subseteq \prec_{sem}$$

We write $\approx_{syn}$, $\approx$ and $\approx_{sem}$ for the equivalence relations induced by the corresponding instance relations (we shall not consider $\prec_{weak}$ any further).

Type systems

We can define atomic type systems corresponding to each instance relation, by exchanging the rule [*sub*] in Figure 2.4 by a suitable rule in each case, as follows:

- The subtyping system $\lambda_{\leq}(\vdash_P)$ is obtained by exchanging the relation $\models$ in rule [*sub*] in Figure 2.4 with the relation $\vdash_P$.
- The subtyping system $\lambda_{\leq}(\models_P)$ is just the system defined by Figure 2.4, taking $\models$ to be $\models_P$.
- The subtyping system $\lambda_{\leq}(\prec_{sem})$ is obtained by exchanging the rule [*sub*] in Figure 2.4 with the rule

$$\frac{C, \Gamma \vdash_P M : \tau \quad C, \Gamma \vdash_P M : \tau \prec_{sem} C', \Gamma' \vdash_P M : \tau'}{C', \Gamma' \vdash_P M : \tau'}$$

The system $\lambda_{\leq}(\prec_{sem})$ collapses the notion of subsumption with the notion of instance (see [73] for further details.) In this system, a principal typing is a minimal judgement with respect to $\prec_{sem}$, *i.e.*, $\mathbf{t}$ is a principal typing judgement for a term M if and only if it holds for any other valid typing judgement $\mathbf{t}'$ for M that we have $\mathbf{t} \prec_{sem} \mathbf{t}'$.

The system $\lambda_{\leq}(\vdash_P)$ is considered together with the instance relation $\prec_{syn}$, the system $\lambda_{\leq}(\models_P)$ is considered together with the instance relation $\prec$, and the system $\lambda_{\leq}(\prec_{sem})$ is considered with the instance relation $\prec_{sem}$. Using this convention, it is easy to see that each notion of instance is indeed *sound* for the corresponding type system, in the sense of Hoang and Mitchell [37]. All these systems type the same pure (constant–free) λ–terms, since they all type exactly the simple typed constant–free terms. This follows immediately from the following theorem, which states that every judgement derivable in the system $\lambda_{\leq}(\prec_{sem})$ has a corresponding derivable judgement in simple types:

Theorem 5.4.1 *For an arbitrary fixed variable α, let the S^{α} denote the map that sends every variable in $\mathcal{V}$ and every constant in P to the variable α. Suppose that M is a constant–free λ–term such that $C, \Gamma \vdash_P M : \tau$ is derivable in the system $\lambda_{\leq}(\prec_{sem})$. Then $S^{\alpha}(\Gamma) \vdash M : S^{\alpha}(\tau)$ is derivable in the simple typed λ–calculus, λ_s.*

PROOF The proof is by induction on the derivation of the judgement $C, \Gamma \vdash_P M : \tau$ in $\lambda_\leq(\prec_{sem})$ (in which the rule [*base*] is not used). In the case of rule [*sub*] (using the relation $\prec_{sem}$), consider the proof step

$$\frac{C, \Gamma \vdash_P M : \tau \quad C, \Gamma \vdash_P M : \tau \prec_{sem} C', \Gamma' \vdash_P M : \tau'}{C', \Gamma' \vdash_P M : \tau'}$$

By induction hypothesis, we have $S^\alpha(\Gamma) \vdash M : S^\alpha(\tau)$ in λ_s. Since $C, \Gamma \vdash_P M : \tau \prec_{sem} C', \Gamma' \vdash_P M : \tau'$, we must have $\Gamma(x)$ and $\Gamma'(x)$ matching for all $x \in \mathcal{D}(\Gamma)$ and τ must match τ'. It follows that $S^\alpha(\Gamma) \subseteq S^\alpha(\Gamma')$ and $S^\alpha(\tau) = S^\alpha(\tau')$, and therefore $S^\alpha(\Gamma') \vdash M : S^\alpha(\tau')$ follows from $S^\alpha(\Gamma) \vdash M : S^\alpha(\tau)$. The remaining cases are obvious and left out. □

Standard procedure for principal typings

It can be shown by well–known methods (induction in typing derivations) that the *standard procedure* (Section 5.1.1) remains valid for generating principal typings in the system $\lambda_\leq(\models_P)$. The proof is analogous to well–known proofs for $\lambda_\leq(\vdash_P)$, see [53, 28]. As for the system $\lambda_\leq(\prec_{sem})$, we can use the property proven by Trifonov and Smith [73], that a principal typing can be obtained by extracting subtyping constraints at every application point in a term (more generally, at every point in a term, where a destructive operation is performed); the resulting principal typing in $\lambda_\leq(\prec_{sem})$ is well–typed in $\lambda_\leq(\vdash_P)$, and by results in [53, 28], we know that an equivalent typing with respect to $\prec_{weak}$ can be obtained by the standard procedure. We will be looking at only a few rather simple terms here, and for these terms it will not be difficult to verify principality of typings in each case.

5.4.2 Exponential lower bound for semantic subtyping

We will show that an exponential lower bound on the dag–size of typings continues to hold for principal typings the system $\lambda_\leq(\prec_{sem})$, under the most powerful instance relation known, $\prec_{sem}$. In the light of this, it appears to be plausible that there may not exist any meaningful instance relation at all, which by–passes the exponential lower bound for atomic subtyping systems.

We will show that there exists a series of terms $\mathbf{Q}_n^{sem}$ such that $\mathbf{Q}_n^{sem}$ has a principal typing $\mathbf{t}_n$ with $2^{\Theta(n)}$ distinct type variables and such that any other typing $\mathbf{t}'_n$ with $\mathbf{t}_n \approx_{sem} \mathbf{t}'_n$ must also contain $2^{\Theta(n)}$ distinct type variables. We assume a non–trivial poset P of base types, containing at least two distinct constants, denoted 0 and 1, with $0 \leq 1$.

There is a natural notion of contextual (observable) equivalence associated with qualified types, introduced in [73]. Let q_1 and q_2 be two qualified types, $q_1 = \tau_1/C_1$ and $q_2 = \tau_2/C_2$. A pair (D, τ) consisting of a constraint set D and a type τ is called a *context* for q_1 and q_2 if and only if D and τ have no variables in common with q_1 or q_2, *i.e.*, $(\mathsf{Var}(D) \cup \mathsf{Var}(\tau)) \cap (\mathsf{Var}(q_1) \cup \mathsf{Var}(q_2)) = \emptyset$. It is then easy, using definitions, to prove the following useful property, which corresponds to the "full type abstraction" theorem in [73] (Theorem 23).

Lemma 5.4.2 *$\tau_1/C_1 \prec_{sem} \tau_2/C_2$ if and only if $C_2 \cup D \cup \{\tau_2 \leq \tau\}$ satisfiable implies $C_1 \cup D \cup \{\tau_1 \leq \tau\}$ satisfiable, for every context (D, τ) for τ_1/C_1 and τ_2/C_2.*

The next definition singles out the set of addresses in a type where a given variable has positive (resp. negative) occurrences:

Definition 5.4.3 If τ is a type expression and α a variable, let $P_\alpha(\tau)$ denote the set of addresses w in $\mathcal{D}(\tau)$ such that $\pi(w) = 0$ and $\tau(w) = \alpha$, and let $N_\alpha(\tau)$ denote the set of addresses w in $\mathcal{D}(\tau)$ such that $\pi(w) = 1$ and $\tau(w) = \alpha$. □

Using the Match Lemma (Lemma 2.3.5) and the fact that valuations are atomic, it is easy to see that $\tau/C \prec_{sem} \tau'/C'$ with C' satisfiable implies that τ and τ' are matching types, with $\mathcal{D}(\tau) = \mathcal{D}(\tau')$. We will tacitly appeal to this fact in the sequel.

Lemma 5.4.4 *Assume $\tau/C \prec_{sem} \tau'/C'$. Let $\alpha \in \mathsf{Var}(\tau)$ with α occurring both negatively and positively in τ. Let $w \in P_\alpha(\tau)$ and $w' \in N_\alpha(\tau)$, and let A^p and A^n be atoms such that $A^p = \tau'(w)$ and $A^n = \tau'(w')$. Then $C' \models_P A^n \leq A^p$.*

PROOF Since $\tau/C \prec_{sem} \tau'/C'$ we have

$$\forall v' \models_P C'.\ \exists v \models_P C.\ \ v(\tau) \leq v'(\tau') \tag{5.1}$$

To show that $C' \models_P A^n \leq A^p$, suppose that $v' \models_P C'$. Then (5.1) shows that the inequality $\tau \leq v'(\tau')$ is satisfiable, by some valuation in P. Since α occurs both positively and negatively in τ, the inequality $\tau \leq v'(\tau')$ implies (by decomposition) the inequalities

$$v'(A^n) \leq \alpha \leq v'(A^p)$$

Since $\tau \leq v'(\tau')$ is satisfiable, it follows that $v'(A^n) \leq_P v'(A^p)$ is true. □

Definition 5.4.5 Let V be a set of variables and C a constraint set. We say that C *differentiates* V if and only if for every pair of distinct variables α and β in V both of the sets $C\{\alpha \mapsto 0, \beta \mapsto 1\}$ and $C\{\alpha \mapsto 1, \beta \mapsto 0\}$ are satisfiable. □

Proposition 5.4.6 *Assume that* $\tau/C \approx_{sem} \tau'/C'$. *Suppose that* $V \subseteq \mathsf{Var}(\tau)$ *is such that:*

1. *Every variable in* V *occurs both positively and negatively in* τ, *and*
2. C *differentiates* V

Then, for any two distinct variables α *and* β *in* V, *and for any* $w \in P_\alpha(\tau)$ *and any* $w' \in P_\beta(\tau)$, *the atoms* $\tau'(w)$ *and* $\tau'(w')$ *are distinct variables in* $\mathsf{Var}(\tau')$. *In particular,* $|V| \leq |\mathsf{Var}(\tau')|$.

PROOF Let α and β be any two distinct variables in V. We have $P_\gamma(\tau)$ and $N_\gamma(\tau)$ non–empty, for $\gamma \in \{\alpha, \beta\}$. Let w and w' be any addresses such that $w \in P_\alpha(\tau)$ and $w' \in P_\beta(\tau)$, and let $A^p_\alpha = \tau'(w)$ and $A^p_\beta = \tau'(w')$. We *claim* that

$$A^p_\alpha \neq A^p_\beta \tag{5.2}$$

To prove (5.2), choose $w_1 \in N_\alpha(\tau)$ and $w_2 \in N_\beta(\tau)$, and let $A^n_\alpha = \tau'(w_1)$ and $A^n_\beta = \tau'(w_2)$. Then Lemma 5.4.4 shows that we have

$$C' \models_P A^n_\alpha \leq A^p_\alpha \tag{5.3}$$

and

$$C' \models_P A^n_\beta \leq A^p_\beta \tag{5.4}$$

Let θ be a renaming of $\tau\{\alpha \mapsto 0, \beta \mapsto 1\}$ with variables distinct from the variables in τ/C and τ'/C'. Then we have

$$\forall w \in P_\alpha(\tau) \cup N_\alpha(\tau).\ \theta(w) = 0$$

and

$$\forall w \in P_\beta(\tau) \cup N_\beta(\tau).\ \theta(w) = 1$$

Since C differentiates V, we know that the set $C \cup \{\tau \leq \theta\}$ is satisfiable. Then, by Lemma 5.4.2, the set $C' \cup \{\tau' \leq \theta\}$ is also satisfiable. But decomposing the inequality $\tau' \leq \theta$, we get that the following inequalities (among

possibly others) must be satisfiable together with C':

$$\begin{aligned} A^p_\alpha &= \tau'(w) \leq \theta(w) = 0 \\ A^p_\beta &= \tau'(w') \leq \theta(w') = 1 \\ A^n_\alpha &= \tau'(w_1) \geq \theta(w_1) = 0 \\ A^n_\beta &= \tau'(w_2) \geq \theta(w_2) = 1 \end{aligned}$$

Now, if our claim (5.2) were false, then we should have $A^p_\alpha = A^p_\beta$ and therefore (by the inequalities above) $A^p_\beta \leq 0$ and $A^n_\beta \geq 1$ would be satisfiable together with C'. But this obviously contradicts (5.4). We must therefore conclude that the claim (5.2) is true.

We have now established the following property:

(∗) For any two distinct variables α and β in V, it is the case that, for any $w \in P_\alpha(\tau)$ and any $w' \in P_\beta(\tau)$, we have $\tau'(w) \neq \tau'(w')$

It is easy to show, by a similar argument, that

(∗∗) For any two distinct variables α and β in V, it is the case that, for any $w \in P_\alpha(\tau)$ and any $w' \in P_\beta(\tau)$, we have $\tau'(w) \in \mathsf{Var}(\tau')$ and $\tau'(w') \in \mathsf{Var}(\tau')$

To see (∗∗), we assume that, say, A^p_α is a constant, 0 or 1. If A^p_α is 0, then we choose θ in such a way that $A^n_\alpha \geq 1$ becomes forced, which contradicts satisfiability with C' because $C' \models_P A^n_\alpha \leq A^p_\alpha$. If $A^p_\alpha = 1$, then we choose θ in such a way that $A^p_\alpha \leq 0$ becomes forced, contradicting satisfiability again.

Now, there is a function f mapping V to addresses such that for each $\alpha \in V$ we have $f(\alpha) \in P_\alpha(\tau)$. Then it follows from (∗) together with (∗∗) that the map $\tau' \circ f$ is an injection from V to $\mathsf{Var}(\tau')$, *i.e.*, for any $\alpha, \beta \in V$ with $\alpha \neq \beta$ we have $\tau'(f(\alpha)) \neq \tau'(f(\beta))$ and $\tau'(f(\alpha)) \in \mathsf{Var}(\tau')$ and $\tau'(f(\beta)) \in \mathsf{Var}(\tau')$. This shows that $|V| \leq |\mathsf{Var}(\tau')|$, and the lemma is proven. □

We will now construct a series of terms $\mathbf{Q}^{sem}_n$ such that the dag–size of principal typings of these terms must grow exponentially, with respect to $\prec_{sem}$. Let

$$\mathbf{D} = \lambda z.\langle z, z\rangle$$

and define as usual $\mathbf{D}^n N$ to be the n–fold application of $\mathbf{D}$ to N. For variables x, y let $P_{x,y} = \langle x, y\rangle$, and define for $n \geq 0$ the terms $\mathbf{Q}^{sem}_n$ by

$$\mathbf{Q}^{sem}_n = \lambda x.\lambda y.\lambda w.\lambda g.\langle\langle g\ (\mathbf{D}^n\ P_{x,y}), g\ w\rangle, P_{x,y}\rangle$$

It is not too difficult to verify (starting with constraint extraction by the *standard procedure* and then applying G- and S–simplification, as usual) that the following completion represents a principal typing for $\mathbf{Q}_n^{sem}$:

$$\lambda x : \alpha.\lambda y : \beta.\lambda w : \tau_w.\lambda g : \tau_w \to \eta.$$
$$\langle\langle g \ \uparrow_{\tau_D}^{\tau_w} (\mathbf{D}^n \ P_{x,y}), g \ w\rangle, P_{x,y}\rangle$$

at the type

$$\tau^{[n]} = \alpha \to \beta \to \tau_w \to (\tau_w \to \eta) \to (\eta \times \eta) \times (\alpha \times \beta) \tag{5.5}$$

and where τ_w has has the shape of T_n (the full binary tree of height n with 2^n leaves) with 2^n distinct variables γ_1 thru γ_{2^n} at the leaves,

$$\tau_w = ((\gamma_1 \times \gamma_2) \times (\gamma_3 \times \gamma_4)) \times \ldots \times ((\gamma_{(2^n-3)} \times \gamma_{(2^n-2)}) \times (\gamma_{(2^n-1)} \times \gamma_{2^n}))$$

and τ_D has the shape of T_n, with leaves alternating between α and β,

$$\tau_D = ((\alpha \times \beta) \times (\alpha \times \beta)) \times \ldots \times ((\alpha \times \beta) \times (\alpha \times \beta))$$

Hence, a principal typing of $\mathbf{Q}_n^{sem}$ has the form

$$C_n, \emptyset \vdash_P \mathbf{Q}_n^{sem} : \tau^{[n]}$$

where C_n, the decomposition of $\tau_D \leq \tau_w$, is just the set

$$C_n = \{\alpha \leq \gamma_i\}_{i \in I} \cup \{\beta \leq \gamma_j\}_{j \in J} \tag{5.6}$$

where I is the set of uneven numbers between 1 and 2^n, and J is the set of even numbers between 1 and 2^n.

The types $\tau^{[n]}$ all have the important property that *every variable occurring in $\tau^{[n]}$ occurs both positively and negatively in the type.* This property allows us to prove:

Theorem 5.4.7 *For any non–trivial poset P of base types, the dag-size of constraint sets as well as of types in atomic principal typings with respect to $\prec_{sem}$ is of the order $2^{\Theta(n)}$ in the worst case.*

PROOF We prove that, over any non–trivial poset P, every principal typing for $\mathbf{Q}_n^{sem}$ with respect to $\prec_{sem}$ has $2^{\Theta(n)}$ distinct type variables in its constraint set as well as in its type.

Let $n \geq 1$ be given; we know that $\mathbf{Q}_n^{sem}$ has a principal typing of the form $C_n, \emptyset \vdash_P \mathbf{Q}_n^{sem} : \tau^{[n]}$, where C_n is given by (5.6) and $\tau^{[n]}$ is given by

(5.5). To simplify notation, let $C = C_n$ and $\tau = \tau^{[n]}$. To fix variable names, write

$$\tau = \alpha \to \beta \to \tau_w \to (\tau_w \to \eta) \to (\eta \times \eta) \times (\alpha \times \beta)$$

where τ_w uses variables γ_1 thru γ_{2^n}. Let $C', \emptyset \vdash_P \mathbf{Q}_n^{sem} : \tau'$ be any other principal typing for $\mathbf{Q}_n^{sem}$. We have $\tau/C \approx_{sem} \tau'/C'$.

With

$$G_\alpha = \{\gamma \mid \alpha \leq \gamma \in C\}$$

and

$$G_\beta = \{\gamma \mid \beta \leq \gamma \in C\}$$

it is clear that C differentiates all variables in $G_\alpha \cup G_\beta$, and hence Proposition 5.4.6 is applicable to the variables in $G_\alpha \cup G_\beta$. It is also clear that C differentiates the set $\{\alpha, \beta\}$, so the proposition applies to this set also. We have

$$C = \{\alpha \leq \gamma \mid \gamma \in G_\alpha \cup G_\beta\}$$

and $|G_\alpha \cup G_\beta| = |\mathsf{Var}(C)| - 2 = |\mathsf{Var}(\tau)| - 3$ (the variable η occurs in τ but not in C). It follows from Proposition 5.4.6 that we have

$$|\mathsf{Var}(\tau')| \geq |\mathsf{Var}(\tau)| - 3 \tag{5.7}$$

We will now show that we also have

$$|\mathsf{Var}(C')| \geq |\mathsf{Var}(C)| - 4 \tag{5.8}$$

To prove (5.8), choose arbitrary $w_\alpha \in N_\alpha(\tau)$ and $w_\beta \in N_\beta(\tau)$ and let

$$\alpha' = \tau'(w_\alpha) \text{ and } \beta' = \tau'(w_\beta)$$

(Proposition 5.4.6 shows that $\tau'(w_\alpha)$ and $\tau'(w_\beta)$ are indeed variables). Define the sets of variables G'_α and G'_β by

$$G'_\alpha = \{\tau'(w) \mid \exists \gamma \in G_\alpha.\ w \in P_\gamma(\tau)\}$$

and

$$G'_\beta = \{\tau'(w) \mid \exists \gamma \in G_\beta.\ w \in P_\gamma(\tau)\}$$

Then Proposition 5.4.6 implies that $|G'_\alpha \cup G'_\beta| \geq |G_\alpha \cup G_\beta| = |\mathsf{Var}(C)| - 2$. Finally, define the sets of variables G''_α and G''_β by

$$G''_\alpha = G'_\alpha \setminus \{\alpha'\} \text{ and } G''_\beta = G'_\beta \setminus \{\beta'\}$$

Then we have

$$|G''_\alpha \cup G''_\beta| \geq |\mathsf{Var}(C)| - 4$$

To show (5.8) it is therefore sufficient to show that we have

$$G''_\alpha \subseteq \mathsf{Var}(C') \tag{5.9}$$

and

$$G''_\beta \subseteq \mathsf{Var}(C') \tag{5.10}$$

We will show how to prove the first inclusion (5.9), the second one follows by the same reasoning.

To prove (5.9), suppose that there exists $\gamma' \in G''_\alpha$ with $\gamma' \notin \mathsf{Var}(C')$, aiming for a contradiction. Then, for some $\gamma \in G_\alpha$ and some $w_\gamma \in P_\gamma(\tau)$, we have

$$\gamma' = \tau'(w_\gamma) \text{ and } \gamma = \tau(w_\gamma)$$

Notice that we have $\pi(w_\gamma) = 0$ and $\pi(w_\alpha) = 1$. Finally, by the definition of G''_α, we have

$$\gamma' \neq \alpha'$$

It is easy to see that we can choose a valuation v_1 satisfying C such that $v_1(\alpha) = 1$. Because $\tau'/C' \prec_{sem} \tau/C$, it follows that there exists a valuation v' satisfying C' such that

$$v'(\tau') \leq v_1(\tau)$$

Since $\pi(w_\alpha) = 1$, this implies that

$$1 = v_1(\alpha) = v_1(\tau)(w_\alpha) \leq v'(\tau')(w_\alpha) = v'(\alpha')$$

so that we have

$$1 \leq v'(\alpha')$$

Let the valuation v'_0 be given by

$$v'_0 = v' \oplus \{\gamma' \mapsto 0\}$$

Because we have assumed that $\gamma' \notin \mathsf{Var}(C')$, we have $v'_0 \models_P C'$ (because $v' \models_P C'$). Moreover, since $\gamma' \neq \alpha'$, we have

$$v'_0(\alpha') = v'(\alpha') \geq 1$$

Now, because $\tau/C \prec_{sem} \tau'/C'$, it follows that there exists a valuation v such that

$$v \models_P C \text{ and } v(\tau) \leq v'_0(\tau') \tag{5.11}$$

Since $\pi(w_\alpha) = 1$ and $\pi(w_\gamma) = 0$, this implies that

$$v(\tau)(w_\alpha) \geq v_0'(\tau')(w_\alpha) \tag{5.12}$$

and

$$v(\tau)(w_\gamma) \leq v_0'(\tau')(w_\gamma) \tag{5.13}$$

But we have

$$v(\tau)(w_\alpha) = v(\alpha) \text{ and } v_0'(\tau')(w_\alpha) = v_0'(\alpha') = 1$$

so therefore (5.12) implies

$$v(\alpha) \geq 1 \tag{5.14}$$

Moreover, we have

$$v(\tau)(w_\gamma) = v(\gamma) \text{ and } v_0'(\tau')(w_\gamma) = v_0'(\gamma') = 0$$

Therefore, (5.13) implies

$$v(\gamma) \leq 0 \tag{5.15}$$

But $\alpha \leq \gamma \in C$, so (5.14) and (5.15) are in contradiction with $v \models C$, obtained in (5.11).

We must conclude that $G''_\alpha \setminus \mathsf{Var}(C') = \emptyset$, so $G''_\alpha \subseteq \mathsf{Var}(C')$, thereby proving (5.9); the property (5.10) is proven analogously. This concludes the proof of (5.8).

The theorem now follows from (5.7) and (5.8), because

$$|\mathsf{Var}(\tau)| \geq |\mathsf{Var}(C)| = 2^n + 2$$

□

5.4.3 Separation

We will now compare the relations $\prec_{syn}$, $\prec$ and $\prec_{sem}$ with respect to the power of the simplifications they validate. We will show that these relations constitute a hierarchy with exponentially large gaps, in the following sense: there is a series of terms such that principal typings with respect to $\prec_{syn}$ are exponentially large, whereas principal typings with respect to $\prec$ are linear; and there is a series of terms such that principal typings with respect to $\prec$ are exponentially large, whereas principal typings with respect to $\prec_{sem}$ are linear.

Separating $\prec_{syn}$ and $\prec$

In the paper [65], we showed an analogue of Theorem 4.2.5 for a system based on $\prec_{syn}$. Here, a weaker form of S–simplification is assumed, based on $\vdash_P$ rather than $\models_P$. We will call this "syntactic S–simplification". Defining $\uparrow^*_C$ and $\downarrow^*_C$ by

$$\uparrow^*_C (A) = \{A' \mid C \vdash_P A \leq A'\}$$

and

$$\downarrow^*_C (A) = \{A' \mid C \vdash_P A' \leq A\}$$

we obtain the syntactic version of S by exchanging the sets $\Uparrow_C$ and $\Downarrow_C$ and the relation $\models_P$ in Definition 4.2.1 with the sets $\uparrow^*_C$ and $\downarrow^*_C$ and the relation $\vdash_P$, respectively. We also use a syntactic notion of cyclic sets, according to which C is cyclic if and only if there are atoms A and A' with $A \neq A'$ such that $C \vdash_P A = A'$. It was shown that, with these syntactic notions of S–simplification and cyclicity, any S–normal form typing with all variables observable and acyclic constraint set is minimal, implying that any other equivalent typing with respect to $\prec_{syn}$ must have at least as many distinct variables in constraint set and type as the minimal one. We can use this result to separate $\prec_{syn}$ from $\prec$, as follows.

Consider a 2–crown of observable variables,

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

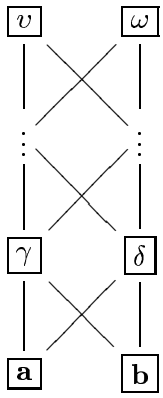
It is easy to verify that, even though all variables in C occur *only positively* in a typing, no S–subsumptions are possible under the *syntactic* notion of S. It then follows from Theorem 5.6 of [65] that a typing with constraints constituting observable crowns is minimal. Compare with Lemma 4.2.2, which showed that such a crown with the bottom variables occurring *negatively* and the top variables occurring *positively* is in S–normal form, with respect to the notion of S–subsumption which uses the relation $\models_P$ rather than $\vdash_P$.

Suppose now that P is the poset with three elements $\{\mathbf{a}, \mathbf{b}, \top\}$ ordered by $\mathbf{a} \leq \top, \mathbf{b} \leq \top$, and suppose that the term language contains a constant a of type $\mathbf{a}$ and a constant b of type $\mathbf{b}$. Consider the series of terms $\mathbf{Q}^{syn}_n$ defined by

$$\mathbf{Q}^{syn}_n = \lambda f_{[n]}.\mathbf{P}^n\langle a, b\rangle$$

where $\mathbf{P}$ and the notation $\lambda f_{[n]} \ldots$ are as defined in Section 5.2. Then, using the analysis of principal typings given in Section 5.2, it is easy to see

that there is a principal typing of $\mathbf{Q}_n^{syn}$ with constraint set C_n consisting of exponentially many crowns of the form



where all variables in the crowns are observable in the typing, and all variables occur only *positively* in the typing. It follows from [65], Theorem 5.6, that the typing is minimal, and that any other principal typing must have exponentially many distinct variables in its constraint set as well as in its type.

However, with respect to $\prec$, all the crowns in these principal typings can be collapsed, because the sets C_n are *cyclic* with respect to $\models_P$ (though not with respect to $\vdash_P$): for every variable γ in C_n, we have $C_n \models_P \gamma = \top$. Therefore, a principle typing under $\prec$ can be given with no type variables at all, and with a type containing only the type $\top$ at the leaves of trees labeled $\times$. As an indication of how this works, consider that the term

$$(\lambda z.\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle z, \langle \pi_2\ z, \pi_1\ z\rangle\rangle\ \mathit{else}\ \langle\langle \pi_2\ z, \pi_1\ z\rangle, z\rangle)\ \langle a, b\rangle$$

has a principal typing with type $(\top \times \top) \times (\top \times \top)$ and empty constraint set. This evidently leads to principal typings for $\mathbf{Q}_n^{syn}$ of linear *dag*–size, with respect to $\prec$.

Separating $\prec$ and $\prec_{sem}$

The relation $\prec_{sem}$ supports simplifications not validated by $\prec$. An interesting case is complete elimination of internal variables over a lattice L of type constants, which, as we know from Example 4.2.3, is not always validated under $\prec$. In contrast, given a qualified type τ/C with internal variables

$\mathsf{Intv}(C)$, we can always find a constraint set C^0 with no internal variables at all, such that $\tau/C \approx_{sem} \tau/C^0$. For let

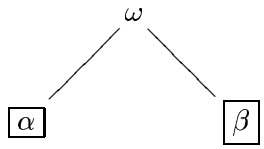
$$C^0 = \{A \leq A' \in \mathsf{Ker}(C) \mid A, A' \in \mathsf{Obv}(C)\}$$

Clearly, then, $\tau/C^0 \prec_{sem} \tau/C$. Conversely, to see $\tau/C \prec_{sem} \tau/C^0$, suppose that $v^0 \models_L C^0$. Let v be given by

$$v = v^0 \oplus \{\alpha \mapsto \bigvee v^0(\downarrow_C^{\mathsf{O}} (\alpha))\}_{\alpha \in \mathsf{Intv}(C)}$$

Here, the function $\downarrow_C^{\mathsf{O}}$ is as defined in Section 4.1.2; the application of v_0 to the set $\downarrow_C^{\mathsf{O}} (\alpha)$ is pointwise. Then one can verify that $v \models_L C$, with the argument given in Section 4.1.2 for the elimination of internal variables using formal joins. Moreover, we have $v(\tau) = v^0(\tau)$, so $v(\tau) \leq v^0(\tau)$ also holds, showing $\tau/C \approx_{sem} \tau/C^0$. Comparing with Section 4.1.2, we see that there is a possible trade–off between either extending the constraint language or extending the instance relation, with respect to enabling more powerful simplifications.

We will now prove that there is an exponential gap between $\prec$ and $\prec_{sem}$, in terms of the simplifications these relations validate. In order to do this, the terms $\mathbf{Q}_n$ from Section 5.2 will be useful. We showed that all principal typings of these terms with respect to $\prec$ have exponentially many variables in constraint sets and types. We will now show that an exponential compression can be gained in principal typings for $\mathbf{Q}_n$, if we simplify under the instance relation $\prec_{sem}$. To see this, recall that principal typings of $\mathbf{Q}_n$ had exponentially many constraint crowns of the form where α and β occur negatively (and not positively) in the typings, and the remaining variables occur positively (and not negatively) in the typings. However, *suppose that P happens to be a lattice L*; then any constraint crown C_k of the form shown in Figure 5.5 can be transformed, under the equivalence $\approx_{sem}$, into the following constraint set C', where ω is a fresh variable:



provided that the *top variables* in C_k (*i.e.*, all variables different from α and β) occur only positively in the typing. Recall that all crowns generated

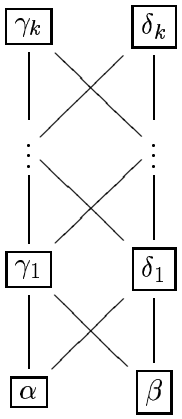


Figure 5.5: Typical constraint crown C_k generated for $\mathbf{Q}_n$

for $\mathbf{Q}_n$ have this form. To see that the transformation mentioned above is sound with respect to $\prec_{sem}$, consider a qualified type τ/C with C containing constraint crowns of the shape of C_k above and variables occurring positively, except α and β sitting at the bottom of the crowns. Let γ_i, δ_i, $i = 1 \ldots k$, be the set of variables occurring at the top of a crown in C, having α and β at the bottom, with all the γ_i and δ_i occurring positively in τ. Then we have

$$S(\tau)/S(C) \prec_{sem} \tau/C$$

with

$$S = \{\gamma_i \mapsto \omega, \delta_i \mapsto \omega\}_{i=1\ldots k}$$

where ω is a fresh variable, occurring nowhere in τ or C. For suppose that $v \models_L C$. Then, taking v' to be the valuation given by

$$v' = v \oplus \{\omega \mapsto v(\alpha) \vee v(\beta)\}$$

then it is easy to see that we have $v' \models_L S(C)$ and $v'(S(\tau')) \leq v(\tau)$. This shows that $\tau/C \approx_{sem} S(\tau)/S(C)$. Applying this transformation exhaustively will effectively map every variable different from α and β to a single, fresh variable ω, representing $\alpha \vee \beta$. Interestingly, this transformation could be validated in a system with explicit join–operations in the constraint language, using the relation $\prec$ (compare with Section 4.1.2).

	$\mathbf{Q}_n^{syn}$	$\mathbf{Q}_n$	$\mathbf{Q}_n^{sem}$
$\prec_{syn}$	2^n	2^n	2^n
$\prec$	n	2^n	2^n
$\prec_{sem}$	n	n	2^n

Figure 5.6: Tight lower bounds for dag–size of principal typings relative to instance relations shown left and witnessed by sequences of terms shown at top

It is straightforward to see that the transformation described above leads to principal typings of $\mathbf{Q}_n$ with respect to $\prec_{sem}$, where all crowns get collapsed into three variables, ω, α and β. As an illustration, consider that the term

$$M = \lambda x.\lambda y.(\lambda z.\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle z, \langle \pi_2\ z, \pi_1\ z\rangle\rangle\ \mathit{else}\ \langle\langle \pi_2\ z, \pi_1\ z\rangle, z\rangle)\ \langle x, y\rangle$$

has the principal typing

$$\{\alpha \leq \omega, \beta \leq \omega\}, \emptyset \vdash_P M : \alpha \to \beta \to (\omega \times \omega) \times (\omega \times \omega)$$

This evidently leads to principal typings for $\mathbf{Q}_n$ of linear *dag*–size, with respect to $\prec_{sem}$.

Summary of results

We can summarize our results about the size–complexity of principal typings in the table shown in figure 5.6. This shows that the series of instance relations

$$\prec_{syn} \subseteq \prec \subseteq \prec_{sem}$$

constitutes a hierarchy separated by exponential gaps with respect to the worst case size of principal typings, thereby showing that the simplifications

validated by each relation are exponentially increasing in strength. While showing that there is potentially a lot to gain by moving to more and more powerful simplifications, the rightmost column of the table shows that, so far, no notion of instance has been proposed which by–passes worst case exponentially sized principal typings. Since $\prec_{sem}$ already appears to exploit the logical structure of subtyping systems to an extreme degree, one may doubt that there exists any sensible notion of instance that would guarantee sub–exponential principal typings. We conjecture that this exponential "degree of freedom" is in fact inherent in subtyping systems.

The features exploited in the series of terms separating the instance relations can be summarized as follows:

$\mathbf{Q}_n^{syn}$: crowns with only positive variables

$\mathbf{Q}_n$: crowns with both positive and negative but no neutral variables

$\mathbf{Q}_n^{sem}$: simple constraints with all variables neutral

Here, a variable is called *neutral* if it occurs both positively and negatively in the typing.

5.5 Linear terms

We know from [37] that subtyping constraints cannot always be eliminated from principal typings in the system $\lambda_{\leq}(\vdash_P)$, relative to any so–called *sound* notion of instance; by the results of [37], series of λ–terms can be defined such that constraint sets must grow at least linearly in the size of the terms, for any principal typing. We have seen in the previous section that the number of constraints may have to grow even exponentially in the system $\lambda_{\leq}(\prec_{sem})$, for any non–trivial P. It is clear that the constructions showing these results must exploit λ–terms that are in some sense "difficult", and there is obviously an interesting relation between the structure of typings and the structure of the λ–terms being typed. It is therefore natural to ask what "difficult" terms look like in general. It may be next to impossible to come up with an exact answer to this question. It seems unlikely that there should be a natural characterization of just those terms that generate large principal typings.

In this section we give a theorem characterizing a very simple set of terms that do *not* require any subtyping constraints at all. The theorem states that every *linear* λ–term has a principal typing, with respect to $\prec_{syn}$,

with empty constraint set. A linear term is one in which every λ–abstracted variable has exactly one free occurrence in the body of the abstraction. This shows that non–linearity is a necessary property of difficult terms, thereby singling out a (no doubt extremely rough) superset of those terms. For these terms, the principal *simple type* (of λ_s) remains principal in the presence of subtyping. Even though a very rough approximation to the real answer, this result is quite useful when one studies difficult typings and the logic of simplifications. The proof will be quite sketchy, since many details can be recovered from standard λ–calculus theory as found in [7, 8, 36].

Theorem 5.5.1 *In the system $\lambda_{\leq}(\vdash_P)$, it holds for every closed linear λ–term M that M has a principal typing with respect to $\prec_{syn}$, containing no subtyping constraints at all.*

PROOF The first step is to realize that every linear term is strongly normalizing with respect to β–reduction (the number of abstractions decrease strictly under β–reduction.) The next step is to show that the *subject expansion* property holds for linear terms in the system $\lambda_{\leq}(\vdash_P)$, *i.e.*, if $C, \Gamma \vdash_P M : \tau$ is derivable and M' is a linear term which β–reduces (in zero or more steps) to M, then $C, \Gamma \vdash_P M' : \tau$ is derivable. (Subject expansion does not, of course, hold for general terms.) One proves the subject expansion property by first proving it for a β–redex, inspecting the form of the subtyping proof for its contraction; one then proves the property for a β–redex in any term context, by induction on the structure of the context; one finally proves general subject expansion by induction in the length of an arbitrary β–reduction starting from a linear term.

The idea now is to prove that every term in β–normal form can be typed using a particularly simple typing proof. Subject expansion then allows us to "pull back" this property to any term.

Recall (from,*e.g.*, [36], p.14) that the set βNF of lambda terms in β-normal form can be defined inductively, as follows:

1. every term variable x is in βNF
2. if $M_1, \ldots, M_n$ are in βNF, then so is $(x\ M_1 \ldots M_n)$ for any variable x
3. if M is in βNF, then so is $\lambda x.M$

Let us say that a subtyping proof of the judgement $C, \Gamma \vdash_P M : \tau$ is *simple*, if every use of the rule [*sub*] is either an application of the rule to a free variable of M or occurs as the last step in the proof. Then one can show, by

induction in the form of a βNF–term, that every linear term in β–normal form has a principal typing with a simple typing proof; showing this is an interesting exercise, and we leave it as such here. It then follows that every *closed* linear term in βNF has a principal typing with no subtyping constraints at all, because (by the previous result) any such term can be typed by a proof which uses the rule [*sub*] only at the last step in the proof; it is easy to see, in turn, that any such application of the rule [*sub*] can be eliminated under $\prec_{syn}$, *i.e.*, if the last step of the proof of the principal typing is

$$\frac{C, \Gamma \vdash_P M : \tau \qquad C \vdash_P \tau \leq \tau'}{C, \Gamma \vdash_P M : \tau'}$$

then the judgement $C, \Gamma \vdash_P M : \tau$ is already principal, with respect to $\prec_{syn}$. We have now established that every closed, linear term in β–normal form has a principal typing with no subtyping assumptions at all. Let M be an arbitrary closed linear term. Let M' be its β–normal form. It has a principal typing with no subtyping assumptions. By subject expansion, so has M. □

Indeed, going back to the "difficult" terms constructed previously in the present chapter, the reader will find that they all exploit non–linearity.

5.6 Typability vs. presentation

The result that principal typings can have exponentially large dag–size shows that, for certain atomic subtyping systems, the *type presentation problem* is strictly harder than the *typability problem*. The problem of type presentation is

- *Given a term M, present its principal typing, if it has one*

and the typability problem is:

- *Given a term M, decide whether it has a typing*

The complexity theoretic separation between these two problems follows from the exponential lower bound result proven here together with previous results in the literature, when we consider subtyping over a lattice of base types. We know from Tiuryn's work [71] (see Theorem 2.4.2) that satisfiability of subtype constraints in finite structural subtyping over a lattice of base types is in PTIME. We also know that deciding typability in the

system $\lambda_{\leq}(\vdash_P)$ of finite, structural subtyping is linear time reducible to the satisfiability problem (see Section 2.4.2). Hence, typability for finite structural subtyping over a lattice is in PTIME. Now, for *atomic* subtyping over a lattice L of base types, typability can be decided in polynomial time, by the previously mentioned results. To see this, recall that, by the *standard procedure* (Section 5.1, see also [53, 28]), one can test typability by the following process:

1. *Constraint extraction*: For a given term M, extract a general (non–atomic) constraint set C_M of size linear in M (by generating a single inequality of the form $\tau \leq \tau'$ at each node of the syntax tree of M)
2. *Expansion*: Expand C_M into atomic form, $C_M^\flat$, by *flattening* C_M (see Lemma 2.3.10)
3. *Satisfiability test*: Test whether $C_M^\flat$ is satisfiable in L

However, since we know from Lemma 2.3.10 that $C_M^\flat$ is satisfiable in L if and only if C_M is satisfiable in $\mathcal{T}_\Sigma^{\mathrm{F}}[s]$, we need not perform the expansion step, if we are only interested in typability. Performing the satisfiability test directly on the set C_M then allows us to test typability in PTIME; indeed, since the expansion step can incur an exponential blow–up in the size of the constraint set, performing the step is potentially catastrophic.

The situation contrasts with the situation for both simple typed λ–calculus and ML, where we can present typings (under dag–representation) within the same time–bound as is required for deciding typability (linear time and exponential time, respectively).

5.7 Non–atomic systems

The results obtained so far may prompt us to question whether atomic subtyping systems are any good at all. We will discuss pro's and con's in this section. In doing so, we will also discuss the question whether non–atomic systems by–pass the exponential degree of freedom found in principal atomic subtypings.

5.7.1 Type size vs. explicitness of information

Non–atomic subtyping systems allow typings of the form $C, \Gamma \vdash_P M : \tau$ with C a general, non–atomic constraint set, containing arbitrary inequalities of

the form $\tau \leq \tau'$. Within such frameworks, one can always find principal typings of size linear in the size of the term. This follows from the well–known fact mentioned in Section 5.6, that principal typings can be generated from a term by extraction of a constant number of constraints at each node in the syntax tree of the term. For structural subtyping systems (see [53, 28]), a typing extracted from a term in this way can be transformed into atomic form by first applying a most general matching substitution to the constraint set and then decomposing the resulting matching set to atomic form (recall the *standard procedure* from Section 5.1). It is the match–step that can cause an exponential blow–up in the size of the typing.

However, avoiding the blow–up in type–size by the use of general inequalities, appears to come at the cost of loss of explicit information content. To see a very simple example of this, suppose we consider typing the term

$$M = \lambda x.\lambda y.\langle x, y\rangle$$

under structural, finite subtyping. A principal completion of M obtained as described above, using general constraints, is

$$\lambda x : \alpha.\ \uparrow^{\delta}_{\beta\times\gamma} \langle \pi_1\ \uparrow^{\beta\times\gamma}_{\alpha} x, \pi_2\ \uparrow^{\beta\times\gamma}_{\alpha} x\rangle$$

resulting in the typing

$$\{\alpha \leq \beta \times \gamma, \beta \times \gamma \leq \delta\}, \emptyset \vdash_P M : \alpha \to \delta$$

It appears to be obvious that this typing is less informative than the principal atomic typing

$$\emptyset, \emptyset \vdash_P M : \alpha \times \beta \to \alpha \times \beta$$

because the latter makes it explicit that a type of pairs is required as argument, and that the term acts as the identity on the argument. To be sure, the latter typing can be obtained from the former (by atomization followed by simplifications), but the point is that the latter typing is only *implicitly* given in the former. Saying, *tout court*, that the former typing already contains the same information as the latter and that's it, is really no different from saying, *e.g.*, that instead of the numerical constant 1 we might as well always write down any mathematically equivalent expression, such as for instance Stirling's formula

$$\lim_{x\to\infty} \frac{\Gamma(x+1)}{(x/e)^x\sqrt{2\pi x}}$$

Most (normal) persons would, in many contexts at least, prefer the constant 1 to this formula, because 1 is a *simplified* form of the formula, in which the complex mathematics needed to compute the simplification has been applied and discarded.

The situation above can be realized in non–structural systems also. All we have to do is to consider constraints of the form

$$\alpha_1 \times \beta_1 \leq \delta \leq \alpha_2 \times \beta_2$$

In any of the tree models considered (see Section 2.1.5), such a constraint is equivalent to the atomic set

$$\{\alpha_1 \leq \delta_1, \beta_1 \leq \delta_2, \delta_1 \leq \alpha_2, \delta_2 \leq \beta_2\}$$

considered over the given tree structure, where δ has been expanded to $\delta_1 \times \delta_2$.

We may summarize by saying that non–atomic systems only by–pass the exponential lower bound by making information implicit which is explicit in the atomic systems. Thus, there seems to be an "exponential trade–off" between type size and explicitness of information: if types are kept small, then explicit information must be lost. Moreover, it is clear that non–atomic systems do not by–pass the exponential degree of freedom found in atomic systems, in the sense that, once the structure of types is made as explicit as possible, there has to be exponentially many distinct variables present in principal typings, in the worst case, under any instance relation suggested so far.

5.7.2 The entailment problem

In many applications it is doubtlessly desirable to avoid expansion of typings. In particular, for systems with PTIME–decidable typability (satisfiability) problems (which include all the lattice–based systems considered in this thesis, see Section 2.4), it seems preferable to avoid atomization when presenting typings, in order to stay within a PTIME–framework. Needless to say, the trade–off depends on the purpose of the type system. If the sole purpose is automatic checking of type soundness, then atomization is definitely not advisable. If a major concern is to give back useful type information to a human reader in small scale systems, then explicitness of information is more of a concern, and one may have a stronger desire to apply atomizing transformations in such a setting. In the setting of polymorphic subtyping,

the trade–off is delicate: on the one hand, we would like to simplify much in order to avoid copying constraint sets with redundant information, but on the other hand we cannot spend too much time doing the simplifications.

In all cases, though, it is an important concern that one should not apply type–expansions unless they are necessary. We have seen extreme cases of this in Section 5.4.3, where we found terms that would yield exponentially large atomic constraint sets, if the atomic typing is generated by the standard procedure, but for which ensuing simplifications can yield exponential succinctness. The simplest example of this is the series $\mathbf{D}^n M$, with $\mathbf{D}$ defined as earlier (Section 5.2), for which naive atomic constraint generation will result in an exponentially sized constraint set, but for which subsequent simplification gives back a typing of linear dag–size. It would obviously be desirable to try to perform the simplifications *without first expanding the constraint set*. With some luck, one might hope to be able to compute the simplified typing in polynomial time. This raises the question:

- *What is the cost of performing simplifications on a general constraint set?*

Another, related concern, is that, even though we do not intend to present typings in completely simplified form, we may want to be able to answer queries about the constraint set. Under this view, a constraint set extracted from a program is a database containing information about the program, from which we may wish to extract information.

The basic computational problem appears in all cases to be the *entailment problem*:

- *Given a constraint set C and types τ and τ', decide whether $C \models \tau \leq \tau'$*

The entailment relation is the natural relation to use, when we wish to query the information contained in a typing, and it is the basic relation involved in the more powerful simplification frameworks of $\prec$ and $\prec_{sem}$. Therefore, we are led to consider the computational complexity of entailment in Part II of the thesis.

Chapter 6

Conclusion to Part I

6.1 Significance of the lower bound

The investigation of the structure of principal typings carried out in Chapter 3, Chapter 4 and Chapter 5 was motivated by a desire to understand the limits of subtype simplification techniques. We have found that equivalence classes of typings with respect to $\prec$ have interesting and useful structure, and that simplification is a confluent process. We used these results to characterize minimal typings, to prove an exponential lower bound on the dag–size of principal typings and to separate a series of instance relations with respect to the simplifications they validate.

The lower bound results do not by themselves allow us to draw definitive conclusions about the practicability of (atomic) subtyping. For example, type inference for ML teaches us to be cautious about jumping to conclusions from theoretical lower bounds. ML has a DEXPTIME-complete type inference problem, and yet this appears to be no problem in actual ML-programs (see [54, Chapter 11.3] with further references.) However, this argument, in its turn, has its limitations too, and it should not be misused to attempt dismissal of all complexity results in the area of programming languages. Moreover, in the case of subtyping, experience definitely shows that subtyping systems are generally very difficult to scale up and that the simplification problem is critical in practice (see references given in Section 1.4).

The construction given in the lower bound proofs do not, of course, explain this observed difficulty in a naively direct way; programs like $\mathbf{Q}_n$ are perhaps not very likely to occur in real systems. However, the proofs do offer an explanation of the logical mechanisms that are responsible for the

complexity of subtyping systems falling within the present frameworks. And certainly, the results show that, for such systems, one will look in vain for simplification procedures that guarantee good (say, polynomial) size worst case behaviour. That this should be so is not obvious a priori.

6.2 Related work

The present lower bound results strengthen the result obtaind by the author in [64], because the present subtyping framework is based on a notion of model–theoretic entailment relations ($\models_P$) rather than the syntactic relation $\vdash_P$ used in [64]. It is easy to see that the present lower bounds imply the result of [64] (and not vice versa).

Most closely related to the present work is that of Hoang and Mitchell [37] and that of Fuh and Mishra [28]. In [37], the authors prove a linear lower bound on the size of principal typings for a whole class of sound instance relations. Our results do not subsume theirs, because we are working with specific notions of instance. However, it may appear difficult to imagine meaningful instance relations considerably stronger than $\prec_{sem}$.

The techniques used by Kanellakis, Mairson and Mitchell in [42] to analyze type size for simply typed lambda calculus and ML are related to our technique for the lower bound proofs in the present chapter, but the main parts of our proofs rely on new techniques tailored for subtyping. The pioneering work by Fuh and Mishra [28] on subtype simplification provided important background in the form of their S- and G-transformations. Fuh and Mishra operate with a notion of minimal typings, but their notion is defined in terms of their transformations, the notion is distinct from ours. The works by Pottier [59], by Trifonov and Smith [73] and by Aiken, Wimmers and Palsberg [4] study entailment based simplifications for polymorphic constrained types. Moreover, the paper by Aiken, Wimmers and Palsberg initiates a systematic study of completeness properties of simplification procedures, which seems closely related in spirit to the investigations made in Part I of this thesis. However, no work apart from [37] appears to have studied the asymptotic behaviour of principal typings with subtyping.

6.3 Open problems

There is much work one could do in continuation of Part I of this thesis, which, admittedly, raises more questions than it answers. One obvious open

problem is to settle the question of type size for principal well–typings relative to any sound notion of instance (see Section 1.5 and [37].) The question is whether the abstract definition of soundness has enough information to allow a proof of a tight result. We believe (see also [68]) that the relation $\prec_{sem}$ must be pretty close to exploiting all relevant logical power of simple subtyping, and based on the lower bound for this relation, it may be doubted that there exists any sound notion of instance at all, which by–passes exponential type size in the worst case.

Another interesting line of research would concern establishing normal forms for typings (in the style of Chapter 3) for more expressive systems over lattices. Such systems would come with strong instance relations or formal lattice–operations (meet and join) in the type- and constraint language or both. We have seen (Theorem 4.1.5) that elimination of internal variables is intractable with respect to $\prec$, but we have also seen that it trivializes over lattices, provided that we have either stronger instance relations such as $\prec_{sem}$ (Section 5.4.3) or we have one of the formal lattice operations (Section 4.1.2). Moreover, it is known [68] (and in any case not too difficult to see, compare also with [46]) that, under sufficiently strong notions of instance, one can get completely rid of constraint sets over a lattice by having *both* of the formal lattice operations (meet and join) in the language. However, this happens at the expense of introducing a much more complicated constraint language (now with an NP–complete satisfiability problem instead of a linear time decidable one, see Section 2.4) and much more complicated types; in such a system, types effectively represent the constraint set by containing meets (in negative positions) and joins (in positive positions). It remains to be seen if any of these systems has particularly good properties, and it really does not appear to be known which (if any) system is the "right" one. Ideal properties are:

1. Tractable simplification problems
2. Highly succinct principal typings
3. Natural canonical (minimal) forms of simplified typings with strong uniqueness properties

Unfortunately, it is to be expected that there is an inherent trade–off here, so that systems with very strong succinctness properties will have (potentially very highly) intractable simplification problems. Part II of this thesis clarifies and confirms this.

Part II

The complexity of subtype entailment over lattices

Chapter 7

Introduction to Part II

The main objective of this part of the thesis is to understand the computational complexity of deciding the *entailment problem* over a variety of structures, in order to determine which properties of subtyping systems contribute to complexity. The entailment problem is:

- *Given a set of subtyping constraints C and type variables α, β, does $C \models \alpha \leq \beta$ hold in the appropriate structure $\mathcal{T}_\Sigma$ of ordered trees?*

We will consider the complexity of the entailment problem in the following variants:

1. *Atomic entailment over a lattice.* Here C is an atomic constraint set, and the entailment relation is $\models_L$, where L is a lattice of constants.
2. *Finite non–structural entailment.* The entailment relation is $\models_\mathcal{M}$ with $\mathcal{M} = \mathcal{T}_\Sigma^\mathrm{F}[n]$.
3. *Non–structural recursive entailment.* The entailment relation is $\models_\mathcal{M}$ with $\mathcal{M} = \mathcal{T}_\Sigma[n]$.
4. *Finite structural entailment over a lattice.* The entailment relation is $\models_\mathcal{M}$ with $\mathcal{M} = \mathcal{T}_\Sigma^\mathrm{F}[s]$.
5. *Structural recursive entailment over a lattice.* The entailment relation is $\models_\mathcal{M}$ with $\mathcal{M} = \mathcal{T}_\Sigma[s]$.

We have already motivated the entailment problem in general, mainly from the perspective of simplification, see in particular Chapter 1 and Section 5.7.2.

The atomic entailment problem is relevant for simplification and query of atomic subtyping systems, including subtype based polymorphic program analysis. We show (Chapter 8) that entailment over a lattice can be solved in linear time. This result is based on a characterization of atomic entailment, which will be used in later chapters.

The non–structural recursive entailment problem is relevant for subtype inference for object oriented programming languages. Thus, the problem is central in the subtype simplification system proposed by Pottier [59], which has an entailment algorithm at its core; however, the algorithm is incomplete, as pointed out by Pottier [59]. The problem also occurs (as a special case) in the simplification framework of Trifonov and Smith [73]. The problem of finding a complete algorithm to decide recursive subtype entailment over non–structurally ordered trees remains open. However, no nontrivial lower bounds on the problem have been given up until now. After some technical preliminaries we prove (Chapter 9) a PSPACE lower bound for this problem, and we show that the problem remains PSPACE-hard even when restricting trees to be finite (non–structural, simple subtyping.) We conjecture that the problem is PSPACE–complete (in PSPACE).

Structural subtyping was introduced by Mitchell [53] and has been used in many subsequent subtyping systems, both with finite and recursive types. We study structural subtyping over a lattice because it is known [71] that even the satisfiability problem is PSPACE-hard for many non–lattices, but in PTIME for large classes of partial orders, including lattices of base types, which are of practical interest (see Section 2.4 for full background). This makes subtyping over lattices of base types particularly interesting from both a practical and a complexity-theoretic point of view.

We begin (Chapter 10) by giving some basic properties for structural recursive subtyping, which will be needed later. We show that *satisfiability* with structural recursive subtyping over a lattice is in PTIME. We then show (Chapter 11) that finite, structural subtype entailment is coNP-complete [33]. We answer (Chapter 12) the corresponding question for *recursive* structural subtype entailment: it is PSPACE-complete. This settles the question of the cost of recursive types for structural subtyping over a lattice of base types:

- For structural subtyping, the addition of recursive types (*i.e.*, the passage from a model of finite trees to a model including infinite trees) has computational cost, unless $NP = PSPACE$.

In contrast, and somewhat surprisingly, our proof of PSPACE-hardness of entailment with non–structural, finite trees leaves open the possibility that, for non–structural subtyping, entailment with finite trees may be as hard as with infinite trees (and we *conjecture* that this is the case).

By our PSPACE-hardness result for finite, non–structural subtyping, we have the to our knowledge first result on the relative complexity of a structural vs. a non–structural theory:

- For finite types, non–structural subtyping is computationally harder than structural subtyping with respect to entailment, assuming NP $\neq$ PSPACE.

Our results are summarized in the following table.

	structural	non–structural
atomic types	$O(n)$	$O(n)$
finite types	$\underline{\text{coNP}}$ coNP	$\underline{\text{PSPACE}}$?
infinite types	$\underline{\text{PSPACE}}$ PSPACE	$\underline{\text{PSPACE}}$?

Here a complexity class above a line indicates a lower bound ("hard for ...") and below a line an upper bound ("contained in ..."). The question marks indicate that no upper bounds for non–structural entailment are known, and this problem remains unsolved at the time of writing. As said, we conjecture that it is in PSPACE and hence PSPACE–complete.

Chapter 8

Atomic entailment

Simple things first. Accordingly, we begin our study of the complexity of subtype entailment with atomic inequalities. Like the problem, the results are simple. But they will be useful later, when problems get less simple.

We consider entailment with atomic inequalities over an arbitrary lattice L. Thus, we are restricting both the constraint language as well as the model. In order to make it clear that the intended model is L, we shall write entailment as $\models_L$. To be precise, we define $C \models_L \alpha \leq \beta$ to hold if and only if we have, for all valuations $v : \mathcal{V} \to L$, that $v \models C$ implies $v \models \alpha \leq \beta$.

8.1 Characterization of atomic entailment

We will prove a characterization theorem for atomic entailment. Recall Theorem 2.4.2 and the definition of the sets $\uparrow_C$ and $\downarrow_C$. Recall also the proof system $\vdash_L$ from Figure 2.3.

Lemma 8.1.1 *Assume C atomic, let $b \in L$, and let α and β be two distinct variables. Assume that*

(i) $b \notin \downarrow_C(\beta)$ *and*

(ii) $C \not\vdash_L \alpha \leq \beta$

Then $\downarrow_C(\beta) = \downarrow_{C\{\alpha \to b\}}(\beta)$

PROOF By Lemma 2.3.14 we have (since $\alpha \neq \beta$) that $C \vdash_L b' \leq \beta$ implies $C[b/\alpha] \vdash_L b' \leq \beta$, for any $b' \in L$, which shows that $\downarrow_C(\beta) \subseteq \downarrow_{C[b/\alpha]}(\beta)$.

To prove the inclusion $\downarrow_{C\{\alpha\mapsto b\}}(\beta) \subseteq \downarrow_C(\beta)$, we use that, whenever $b' \in \downarrow_{C\{\alpha\mapsto b\}}(\beta)$, there must exist a path $P_{b'}$ in $C\{\alpha \mapsto b\} \cup L$ (regarded as a digraph) witnessing this fact. The inclusion then follows from the property

(*) For any $b' \in L$ and any path $P_{b'}$ in $C\{\alpha \mapsto b\} \cup L$ witnessing $b' \in \downarrow_{C\{\alpha\mapsto b\}}(\beta)$ there is a path P' in $C \cup L$ witnessing $b' \in \downarrow_C(\beta)$.

We proceed to prove (*) by induction on the length $|P_{b'}|$ of a path $P_{b'}$ in $C\{\alpha \mapsto b\} \cup L$ of shortest length witnessing $b' \in \downarrow_{C\{\alpha\mapsto b\}}(\beta)$. In case $|P_{b'}| = 0$, we have $b' = \beta$, which is impossible, because b' is a constant and β is a variable. For the inductive case with $|P_{b'}| > 0$, let $P_{b'}$ be the path

$$P_{b'} = b' \leq A_1 \leq A_2 \leq \ldots \leq A_k \leq \beta$$

We proceed by cases on the form of $P_{b'}$.

Case 1. Suppose first that b' is the only constant on the path, so we have $A_i = \gamma_i$ for some variables γ_i, $i = 1 \ldots k$. Then, since $P_{b'}$ is in $C\{\alpha \mapsto b\} \cup L$, we have $\gamma_i \neq \alpha$ for $i = 1 \ldots k$. Therefore the path

$$\gamma_1 \leq \gamma_2 \leq \cdots \leq \gamma_k \leq \beta$$

is in $C \cup L$. Hence, to show $b' \in \downarrow_C(\beta)$ it suffices to show that $b' \leq \gamma_1 \in C \cup L$. Suppose it is not. Then $\alpha \leq \gamma_1$ must be in $C \cup L$ with $b' = b$, contradicting the assumptions (i) and (ii), and hence $b' \in \downarrow_C(\beta)$ in this case.

Case 2. Suppose next that at least one of the A_i is some constant $b'' \in L$. Let $A_m = b''$, $1 \leq m \leq k$, where we can assume w.l.o.g. that m is the least index such that A_m is a constant, so for $i = 1 \ldots m-1$ we have A_i a variable γ_i. The path $P_{b'}$ now looks as follows

$$P_{b'} = b' \leq \gamma_1 \leq \ldots \leq \gamma_{m-1} \leq b'' \leq A_{m+1} \leq \ldots \leq A_k \leq \beta$$

Then we have $b'' \in \downarrow_{C\{\alpha\mapsto b\}}(\beta)$ by a path strictly smaller than $P_{b'}$, and so, by induction hypothesis, we have $b'' \in \downarrow_C(\beta)$. It is therefore sufficient to show that $b' \leq b'' \in (C \cup L)^*$. We proceed to show this by analysis of sub-cases.

Case 2.1. First consider the case where $m = 1$, so we have $b' \leq b'' \in C\{\alpha \mapsto b\} \cup L$. If $b' \leq b'' \notin C \cup L$, then we must have either (1) $b'' = b'$, or (2) $\alpha \leq b'' \in C \cup L$ with $b' = b$, or (3) $b' \leq \alpha \in C \cup L$ with $b'' = b$. In case (1) we get $b' \leq b'' \in (C \cup L)^*$ as desired. In case (2) we get $C \vdash_L \alpha \leq \beta$ from $b'' \in \downarrow_C(\beta)$ (which holds by induction), contradicting assumption (ii), so this case is impossible. In case (3) we get $b \in \downarrow_C(\beta)$ from $b'' \in \downarrow_C(\beta)$,

contradicting assumption (i), so this case is also impossible. This shows that $b' \leq b'' \in (C \cup L)^*$ holds in the case where $m = 0$.

Case 2.2. Next consider the case where $m > 1$. Notice that none of the γ_i $(1 \leq i \leq m-1)$ can be α, and so $\gamma_j \leq \gamma_{j+1} \in C \cup L$ for $j = 1 \ldots m-2$. Therefore, it is sufficient to show that (1) $\gamma_{m-1} \leq b'' \in C \cup L$ and (2) $b' \leq \gamma_1 \in C \cup L$. As for (1), if $\gamma_{m-1} \leq b'' \notin C \cup L$, then (since $\gamma_{m-1} \neq \alpha$) we must have $\gamma_{m-1} \leq \alpha \in C \cup L$ with $b'' = b$, which entails $b \in{\downarrow_C}(\beta)$ (since we have $b'' \in{\downarrow_C}(\beta)$ by induction), contradicting assumption (i). So we must conclude that (1) holds. As for (2), if $b' \leq \gamma_1 \notin C \cup L$, then (since $\gamma_1 \neq \alpha$) we must have $\alpha \leq \gamma_1 \in C \cup L$ with $b' = b$. But then, by (1) together with $\gamma_i \leq \gamma_{i+1} \in C \cup L$, $1 \leq i \leq m-2$, and $b'' \in{\downarrow_C}(\beta)$, we get that $C \vdash_L \alpha \leq \beta$, in contradiction with assumption (ii). Therefore, we must conclude that (2) also holds, thereby proving $b' \leq b'' \in (C \cup L)^*$, and the proof of $(*)$ is finished. □

We can now prove the main result of this chapter (Theorem 8.1.2 below). For the later purposes, it will be convenient to consider entailment with infinite atomic constraint sets.

Theorem 8.1.2 *Let C be a possibly infinite, atomic constraint set, which is satisfiable in a complete lattice L of constants. Then $C \models_L \alpha \leq \beta$ for distinct variables α and β if and only if one of the following conditions is satisfied:*

(i) $C \vdash_L \alpha \leq \beta$

(ii) $\bigwedge {\uparrow_C}(\alpha) \leq_L \bigvee {\downarrow_C}(\beta)$

PROOF $(\Rightarrow)$. Assume $C \models_L \alpha \leq \beta$ and $C \nvdash_L \alpha \leq \beta$. We must then show

$$\bigwedge {\uparrow_C}(\alpha) \leq_L \bigvee {\downarrow_C}(\beta) \tag{8.1}$$

We proceed by contradiction, assuming

$$\bigwedge {\uparrow_C}(\alpha) \not\leq_L \bigvee {\downarrow_C}(\beta) \tag{8.2}$$

Since $C \models_L \alpha \leq \beta$, we have by substitutivity of $\models_L$ (Lemma 2.3.14) and $\alpha \neq \beta$ that

$$C\{\alpha \mapsto \bigwedge {\uparrow_C}(\alpha)\} \models_L \bigwedge {\uparrow_C}(\alpha) \leq \beta \tag{8.3}$$

Let $C^\dagger = C\{\alpha \mapsto \bigwedge \uparrow_C(\alpha)\}$. By the characterization of satisfiability (Theorem 2.4.1) we know, since C is consistent, that the map

$$\{\alpha \mapsto \bigwedge \uparrow_C(\alpha)\}$$

can be extended to a satisfying valuation for C. It follows that $C^\dagger$ is consistent. Then, by Theorem 2.4.1 again, we get that the valuation

$$\{\gamma \mapsto \bigvee \downarrow_{C^\dagger}(\gamma)\}_{\gamma \in \mathsf{Var}(C^\dagger)}$$

satisfies $C^\dagger$. By (8.3) we then have

$$\bigwedge \uparrow_C(\alpha) \leq_L \bigvee \downarrow_{C^\dagger}(\beta) \tag{8.4}$$

Now, if $C \models_L \bigwedge \uparrow_C(\alpha) \leq \beta$ were the case, then Theorem 2.4.1 would entail $\bigwedge \uparrow_C(\alpha) \leq_L \bigvee \downarrow_C(\beta)$ via the satisfying map $\{\gamma \mapsto \bigvee \downarrow_C(\gamma)\}_{\gamma \in \mathsf{Var}(C)}$, contradicting (8.2). We must therefore conclude that

$$C \not\models_L \bigwedge \uparrow_C(\alpha) \leq \beta$$

Hence we must have

$$\bigwedge \uparrow_C(\alpha) \notin \downarrow_C(\beta) \tag{8.5}$$

But (8.5) together with the assumption that $C \not\models_L \alpha \leq \beta$ allows us to apply Lemma 8.1.1, which shows that

$$\downarrow_C(\beta) = \downarrow_{C^\dagger}(\beta) \tag{8.6}$$

Composing (8.6) with (8.4) we get

$$\bigwedge \uparrow_C(\alpha) \leq_L \bigvee \downarrow_C(\beta) \tag{8.7}$$

But (8.7) contradicts (8.2), thereby proving (8.1) and hence the implication ($\Rightarrow$).

($\Leftarrow$). If (i) is the case, the result follows immediately, and if (ii) is the case, the result follows because we evidently have

$$C \models_L \alpha \leq \bigwedge \uparrow_C(\alpha) \text{ and } C \models_L \bigvee \downarrow_C(\beta) \leq \beta$$

□

We will now draw some direct consequences from Theorem 8.1.2. Theorem 8.1.2 and the method of its proof shows that, whenever we have $C \not\models_L \alpha \leq \beta$ with C satisfiable, then the map

$$\{\alpha \mapsto \bigwedge \uparrow_C(\alpha), \beta \mapsto \bigvee \downarrow_C(\beta)\}$$

is a witness to this fact:

Corollary 8.1.3 *Assume that C is a consistent atomic constraint set. If $C \not\models_L \alpha \leq \beta$, then one has*

1. *$C\{\alpha \mapsto \bigwedge \uparrow_C(\alpha), \beta \mapsto \bigvee \downarrow_C(\beta)\}$ is consistent, and*
2. *$\bigwedge \uparrow_C(\alpha) \not\leq_L \bigvee \downarrow_C(\beta)$*

PROOF By the assumption $C \not\models_L \alpha \leq \beta$, Theorem 8.1.2 implies that we have

$$C \not\vdash_L \alpha \leq \beta \tag{8.8}$$

and

$$\bigwedge \uparrow_C(\alpha) \not\leq_L \bigvee \downarrow_C(\beta) \tag{8.9}$$

From (8.9) we get (by the argument used in the proof of Theorem 8.1.2) that

$$\bigwedge \uparrow_C(\alpha) \notin \downarrow_C(\beta) \tag{8.10}$$

From (8.8) and (8.10) one then gets (by the argument used in the proof of Theorem 8.1.2) that $\downarrow_C(\beta) = \downarrow_{C^\dagger}(\beta)$ with $C^\dagger = C\{\alpha \mapsto \bigwedge \uparrow_C(\alpha)\}$. It then follows, as in the proof of Theorem 8.1.2, that $C\{\alpha \mapsto \bigwedge \uparrow_C(\alpha), \beta \mapsto \bigvee \downarrow_C(\beta)\}$ is consistent. □

Theorem 8.1.2 shows that we get a sound and complete axiomatization of the relation $\models_L$ on *atomic* constraint sets by adding the following rules to $\vdash_L$:

$$\frac{C \vdash_L A \leq b_1 \quad C \vdash_L A \leq b_2}{C \vdash_L A \leq b_1 \wedge_L b_2}$$

$$\frac{C \vdash_L b_1 \leq A \quad C \vdash_L b_2 \leq A}{C \vdash_L b_1 \vee_L b_2 \leq A}$$

$$\frac{C \vdash_L b_1 \leq b_2 \quad b_1 \not\leq_L b_2}{C \vdash_L \tau \leq \tau'}$$

The theorem also shows that the predicate $C \models_L \alpha \leq \beta$, where C is atomic, can be decided very efficiently; this is discussed in the following section.

8.2 Linear time decidability of atomic entailment

Theorem 8.1.2 immediately yields a linear time decision procedure for atomic entailment over a complete lattice, because the theorem reduces entailment to a *single–source reachability* problem:

Corollary 8.2.1 *Assuming $\vee_L, \wedge_L$ and $\leq_L$ are constant-time operations, given consistent constraint set C of size n (number of symbols in C) it can be decided in linear time whether $C \models_L \alpha \leq \beta$.*

PROOF Note that C, viewed as a directed graph, has $O(n)$ vertices and $O(n)$ edges. Criterion (i) can be decided in time $O(n)$ by computing the set of of nodes reachable from α in C. As for criterion (ii), both $\bigwedge \uparrow_C(\alpha)$ and $\bigvee \downarrow_C(\beta)$ can be computed in time $O(n)$, again by computing the set of nodes reachable from α in C and by computing the set of nodes reachable from β along the reverse edges in C. □

It is not surprising that atomic entailment over a *finite* lattice should be in PTIME, by a naive algorithm. This is so, because we can represent negation by explicit enumeration, in a finite lattice. We can therefore realize the standard logical reduction of entailment to unsatisfiability, and since satisfiability is in PTIME, the result follows. In full detail, the naive PTIME algorithm for deciding the entailment $C \models_L \alpha \leq \beta$ is just this:

```
for (b1, b2) ∈ L × L do
    if b1 ⋠L b2 then
      C' := C{α ↦ b1, β ↦ b2};
      if C' is satisfiable then return NO;
(* end for *)
return YES;
```

This naive algorithm, apart from being restricted to work on a finite lattice, is quadratically dependent upon the size of the lattice, which the algorithm implicit in Theorem 8.1.2 is not. Moreover, the repeated calls to the satisfiability test is a substantial additional overhead in the naive algorithm. It appears to be intuitively obvious that the algorithm of Theorem 8.1.2 is optimal, and that it yields a very fast procedure to decide atomic entailment in practice.

8.3 Representation

So far we have restricted the constraint language as well as the model, fixing the model to be L. It makes sense, however, to consider the language of atomic constraints over $\mathcal{T}_\Sigma$ as well. In this section, we consider what happens when we choose structurally ordered trees, so consider $\mathcal{T}_\Sigma$ to be that structure for now. We focus on this case, because it will be important for

our understanding of entailment with general constraints over structurally ordered trees, and the results obtained here will be used later.

Exchanging the model L with $\mathcal{T}_\Sigma$ for atomic constraints makes a difference. To see this, consider the following example:

Example 8.3.1 Let C be the atomic constraint set $C = \{\top \leq \beta\}$, and consider the query $C \models_L \alpha \leq \beta$? This entailment obviously holds in L (even though α is completely unconstrained) and the explanation, in terms of Theorem 8.1.2, is that we have

$$\bigwedge \uparrow_C(\alpha) = \top = \bigvee \downarrow_C(\beta)$$

However, the entailment $C \models \alpha \leq \beta$ clearly does not hold in $\mathcal{T}_\Sigma$, because a satisfying valuation could map α to any structured tree (such as, *e.g.*, $\top \times \top$.) This means that the characterization in Theorem 8.1.2 is invalid for $\mathcal{T}_\Sigma$ ordered structurally. □

This example is quite pathological, exploiting that one of the variables (α) is unconstrained and the other (β) is not. As we will show now, this is essentially the only way such examples can be produced, and Theorem 8.1.2 will therefore remain a useful tool for understanding atomic entailment over $\mathcal{T}_\Sigma$. This is not surprising; atomic constraints cannot talk about structured trees in anything but a trivial way, since such constraints cannot mention tree structure explicitly. We will now make this observation technically, for the record.

Definition 8.3.2 Given constraint set C and variable α, we say that α is *equivalent to a constant* in C, if and only if we have

$$\forall v : \mathcal{V} \to \mathcal{T}_\Sigma.\ v \models C \Rightarrow v(\alpha) \in L$$

i.e., all solutions to C must map α to a constant. □

The following lemma states that, for atomic sets, satisfiability and (under certain conditions) entailment over the model $\mathcal{T}_\Sigma$ can be faithfully represented in the model L. The lemma shows that $C \models_{\mathcal{T}_\Sigma} \alpha \leq \beta$ is equivalent to $C \models_L \alpha \leq \beta$ except for the pathological special case where one of the variables is (indirectly) constrained by elements of L and the other is not. The lemma, while not very surprising, will be useful to have later, when we will consider entailment over $\mathcal{T}_\Sigma$; there also, we shall need to sort out pathological special cases, and in that setting there will be more of them. The lemma below will help us do so.

Lemma 8.3.3 *(Representation in L) Let C be a possibly infinite, atomic constraint set.*

1. *If $v \models_{\mathcal{T}_\Sigma} C$, then $\widehat{v} \models_L C$ where $\widehat{v} : \mathcal{V} \to L$ is given by*

$$\widehat{v}(\alpha) = \begin{cases} v(\alpha) & \text{if } v(\alpha) \in L \\ \widehat{b} & \text{if } v(\alpha) \notin L \end{cases}$$

   where $\widehat{b}$ is an arbitrary, fixed constant in L. In particular, C is satisfiable in $\mathcal{T}_\Sigma$ if and only if C is satisfiable in L.

2. *Let C be satisfiable in $\mathcal{T}_\Sigma$. Then*

   (a) *if neither α nor β are equivalent to a constant in C, then*

$$C \models_{\mathcal{T}_\Sigma} \alpha \leq \beta \text{ if and only if } C \models_L \alpha \leq \beta$$

   (b) *if both α and β are equivalent to a constant in C, then*

$$C \models_{\mathcal{T}_\Sigma} \alpha \leq \beta \text{ if and only if } C \models_L \alpha \leq \beta$$

   (c) *if one but not the other of α and β is equivalent to a constant in C, then $C \not\models_{\mathcal{T}_\Sigma} \alpha \leq \beta$*

PROOF To prove the first claim, assume v satisfies C in $\mathcal{T}_\Sigma$. Then $\widehat{v}$ satisfies every inequality in C:

- If $\alpha \leq \beta \in C$, then we have $v(\alpha)$ and $v(\beta)$ matching (by Lemma 2.3.5), hence either $\widehat{v}(\alpha) = v(\alpha)$ and $\widehat{v}(\beta) = v(\beta)$, or else $\widehat{v}(\alpha) = \widehat{v}(\beta) = \widehat{b}$. In either case, the inequality is satisfied by $\widehat{v}$.
- If $\alpha \leq b \in C$, $b \in L$, then (by Lemma 2.3.5) $v(\alpha) \in L$, hence $\widehat{v}(\alpha) = v(\alpha)$, and $\widehat{v}$ satisfies the inequality.

Remaining cases are symmetric and left out. This proves the first part of the lemma.

We now prove the second claim of the lemma. The implication $C \models_{\mathcal{T}_\Sigma} \alpha \leq \beta \Rightarrow C \models_L \alpha \leq \beta$ is obvious in all cases. To see the opposite implication in the first sub-claim, assume that neither α nor β is equivalent to a constant in C, and assume $C \models_L \alpha \leq \beta$. Since neither α nor β is equivalent to a constant in C, we have

$$\uparrow_C (\alpha) = \downarrow_C (\beta) = \emptyset$$

and hence

$$\bigwedge \uparrow_C (\alpha) = \top \not\leq_L \bot = \bigvee \uparrow_C (\beta) \tag{8.11}$$

Since C is satisfiable in L, by the first claim of the present lemma, we can apply Theorem 8.1.2, which together with (8.11) shows that $C \vdash_L \alpha \leq \beta$, end therefore obviously $C \models_{\mathcal{T}_\Sigma} \alpha \leq \beta$ must hold.

To prove the second sub-claim, assume that both α and β are equivalent to a constant in C, and assume $C \models_L \alpha \leq \beta$. Let $v \models_{\mathcal{T}_\Sigma} C$; then $\widehat{v} = v$, since v must map both α and β to a constant in L. By the first claim of this lemma, we have $\widehat{v} \models_L C$, hence $v \models_L C$, and so $v \models_L \alpha \leq \beta$, hence also $v \models_{\mathcal{T}_\Sigma} \alpha \leq \beta$. We have now shown that $C \models_L \alpha \leq \beta$ implies $C \models_{\mathcal{T}_\Sigma} \alpha \leq \beta$ in this case also.

To see the third sub-claim, if one, say α, but not the other, say β, is equivalent to a constant in C, then there is a satisfying valuation v mapping β to a non–constant and α to a constant; then $v(\alpha) \not\leq v(\beta)$. □

Chapter 9

Non–structural entailment

In this chapter we consider entailment over non–structural trees, so we fix the ordered structure $\mathcal{T}_\Sigma$ to be $\mathcal{T}_\Sigma[n]$, *i.e.*, trees over $\Sigma = \{\top, \bot, \times, \rightarrow\}$ ordered non–structurally, where Σ is the lattice shown in Figure 2.1. Section 9.1 contains various technical preliminaries, mainly related to connections between constraints and automata, which we will be using heavily later. Section 9.2 and Section 9.3 establish that recursive non–structural entailment is PSPACE–hard. Section 9.4 establishes PSPACE–hardness for the finite case. We leave the upper bound question open; we conjecture that the problem is in PSPACE, both in the finite and in the infinite case.

9.1 Constraint graphs and constraint automata

To understand the complexity of the entailment problem over trees, we shall find it useful to consider constraint sets as non–deterministic finite automata. This section gives several technical preliminaries related to that.

We will describe two ways of constructing finite automata from constraint sets. The first construction amounts to the observation that a constraint set can be regarded as a non-deterministic finite automaton, when inequalities are viewed as ϵ-transitions and constructed types have transitions on elements of $\mathbf{A}$. The second construction is essentially a determinization of the first one, by the standard subset construction; it was used in [57] and [59] to show that any consistent constraint set has a solution (recall the notion of consistency for non–structural sets, Definition 2.3.3).

9.1.1 Constraint graph

For a given constraint set C, we define a labeled digraph, called the *constraint graph* associated with C and denote it $\mathcal{G}_C = (V, E)$. The vertex set V is the set of subterms occurring in C. The edge relation E in $\mathcal{G}_C$ is a set of edges labeled with elements of $\mathbf{A} \cup \{\epsilon\}$; an edge from τ to τ' labeled with $a \in \mathbf{A} \cup \{\epsilon\}$ is written $\tau \mapsto_a \tau'$, for $\tau, \tau' \in V$. The set E of labeled edges is defined as the least relation $E \subseteq V \times (\mathbf{A} \cup \{\epsilon\}) \times V$ satisfying the following conditions:

- For all $\tau \in V$, $\tau \mapsto_\epsilon \tau$
- If $\tau \leq \tau' \in C$, then $\tau \mapsto_\epsilon \tau'$
- If $\tau = \tau_1 \times \tau_2$ is in V, then $\tau \mapsto_f \tau_1$ and $\tau \mapsto_s \tau_2$
- If $\tau = \tau_1 \to \tau_2$ is in V, then $\tau \mapsto_d \tau_1$ and $\tau \mapsto_r \tau_2$

The relation $\mapsto_\epsilon$ represents (the reflexive closure of) the inequalities in C, by the first two conditions. The next two conditions represent the syntactic structure of terms by the relations $\mapsto_a$, $a \in \mathbf{A}$. It is clear that $\mathcal{G}_C$ can be constructed from C in polynomial time.

Drawing constraint graphs

In order to have a convenient way of drawing constraint graphs, we will identify a term τ in V with a unique copy of its main constructor. In order to avoid cluttering of drawings, we shall sometimes use one or both of these conventions when drawing constraint graphs:

1. We shall normally not show reflexive edges (ϵ-loops).
2. Often we shall not decorate ϵ-edges explicitly as such – edges without labels are to be taken as ϵ-edges.

A simple example of a constraint graph follows.

Example 9.1.1 (Drawing a constraint graph)
Consider as an example the constraint set

$$C = \{\alpha \times \beta \leq \beta \times \gamma, \beta \times \gamma \leq \delta \times \eta\}$$

The constraint graph $\mathcal{G}_C$ is shown in Figure 9.1. □

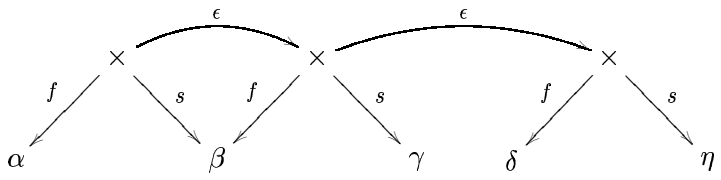


Figure 9.1: Constraint graph $\mathcal{G}_C$, Example 9.1.1

9.1.2 Constraint graph closure

Figure 9.2 shows the three closure rules (transitive closure, product decomposition, arrow decomposition) defining $\mathsf{Cl}(C)$ in the form of rules for edge–additions to $\mathcal{G}_C$. The closure rules add the dashed ϵ-edges to the graph, provided the solid edges are already present in the graph. The closure $\mathsf{Cl}(\mathcal{G}_C)$ is the least graph containing $\mathcal{G}_C$ and closed under the rules of Figure 9.2. It is obvious that $\mathsf{Cl}(\mathcal{G}_C)$ represents $\mathcal{G}_{\mathsf{Cl}(C)}$.

We give an example of constraint graph closure, continuing the example constraint set and constraint graph from Example 9.1.1.

Example 9.1.2 (Constraint graph closure)
Figure 9.3 outlines the closure of $\mathcal{G}_C$ from Figure 9.1 for the set C of Example 9.1.1.

Edges added under closure are shown as dashed edges. We have not shown reflexive edges, and to avoid clutter, not all transitive edges added are shown in the figure; the transitive edges $\alpha \mapsto_\epsilon \gamma$, $\alpha \mapsto_\epsilon \delta$ and $\beta \mapsto_\epsilon \eta$ are left out in the figure. □

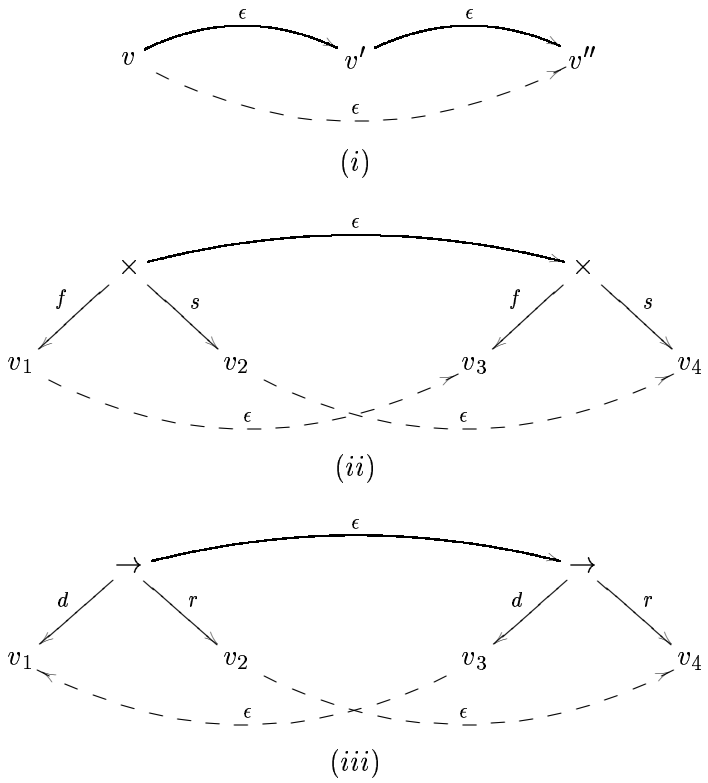


Figure 9.2: Constraint graph closure rules: (i) transitive closure, (ii) product decomposition and (iii) arrow decomposition. Given the solid arcs, the closure rules add the dashed arcs.

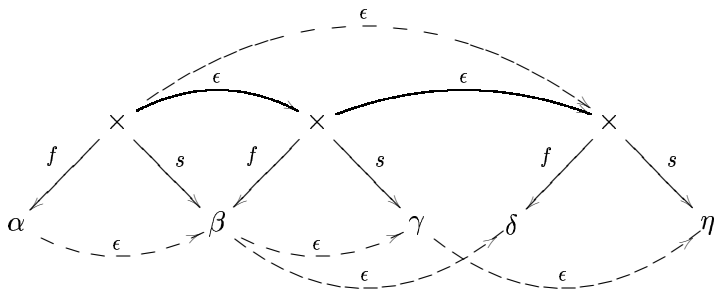


Figure 9.3: Closure (partial) of $\mathcal{G}_C$, Example 9.1.1

9.1.3 Non–deterministic constraint automaton

The constraint graph $\mathcal{G}_C$ can be directly regarded as a nondeterministic finite automaton (NFA) over the alphabet $\mathbf{A}$, where the labeled edge relation E defines the transition relation δ_N. The *nondeterministic constraint automaton* (NCA for short) associated with a set C and with a specified start state q_0, is denoted $\mathcal{A}_N^C(q_0) = (Q, \mathbf{A}, \delta_N, q_0, Q)$. The state set Q is V; the alphabet Σ is $\mathbf{A}$; every state is an accept state.

The iterated transition relation of an NFA, denoted $\widehat{\delta}_N$, is defined in the standard way (see [38]). We sometimes write $q \mapsto^w q'$ to denote a w-transition from q to q' in the automaton, *i.e.*, $q' \in \delta_N(q, w)$.

Notice that the transition relation $\mapsto^a$ of the automaton is different from the relations $\mapsto_a$ of the constraint graph. For instance, if $\tau \mapsto_a \tau'$ and $\tau' \mapsto_\epsilon \tau''$, then $\tau \mapsto^a \tau''$ is a transition in the automaton, but $\tau \mapsto_a \tau''$ is not a single edge in the constraint graph.

9.1.4 Deterministic constraint automaton

We shall now define a *deterministic constraint automaton* (DCA for short) associated with a *non–structural* constraint set C, along the lines of [57, 59]. We shall later modify the construction to encompass structural constraint sets as well.

We shall assume here that constraint sets are given in a special form.

Definition 9.1.3 Call a set C *simple*, if for every inequality $\tau \leq \tau'$ in C both τ and τ' have depth at most 1, *i.e.*, every term is either an atom

(variable or constant) or a binary constructor applied to atoms. □

It is easy to convert any constraint set into simple form, by a linear time transformation which recursively substitutes a term of the form, *e.g.*, $\tau \times \tau'$ by the set $\{\alpha \times \beta, \alpha = \tau, \beta = \tau'\}$.

We define a DCA associated with a *non–structural, simple and closed* constraint set C. The automaton can be constructed from any given non–structural constraint set, provided the set is transformed into simple form and then closed (the closure of a simple set is again simple.) In the sequel, we assume that this has been done.

Let $\mathsf{Nvt}(C)$ be the set of subterms occurring in C which are not variables and not $\top$. For a given type τ and variable α, the set of terms $\Uparrow_C(\tau)$ is defined by setting

$$\Uparrow_C(\alpha) = \{\tau \in \mathsf{Nvt}(C) \mid \alpha \leq \tau \in C\}$$

For non-variable terms τ we set $\Uparrow_C(\tau) = \{\tau\}$. We now define a DFA constructed from C, with designated start state q_0, in the style of [57, 59]; the DFA is called $\mathcal{A}_D^C(q_0)$ and is of the form $(Q, \Sigma, q_0, \delta_D)$ with state set Q given by $Q = \wp(\mathsf{Nvt}(C))$, the set of subsets of $\mathsf{Nvt}(C)$, alphabet $\Sigma = \mathbf{A}$, initial state $q_0 \in Q$, transition function δ_D given by:

- For $q = \{\alpha_i \to \alpha_i'\}_i$:

$$\delta_D(q, d) = \bigcap_i \Uparrow_C(\alpha_i)$$
$$\delta_D(q, r) = \bigcup_i \Uparrow_C(\alpha_i')$$

- For $q = \{\alpha_i \times \alpha_i'\}_i$:

$$\delta_D(q, f) = \bigcup_i \Uparrow_C(\alpha_i)$$
$$\delta_D(q, s) = \bigcup_i \Uparrow_C(\alpha_i')$$

- For q not of the forms above, the transitions functions are undefined.

9.1.5 Term automata

As is done in [57, 59] for non–structural subtyping, we can use deterministic constraint automata to define regular trees presented as *term automata*, in the style of [45]. To do so, we need to provide a *labeling function* (see [45])

$$\ell : Q \to \Sigma$$

With $\mathsf{Con}(\tau) \in \Sigma$ denoting the main constructor of a constructed type τ, we define for $q \in Q$ the set $\mathsf{Con}(q) = \{\mathsf{Con}(\tau) \mid \tau \in q\}$. Then we define the labeling function $\ell_\wedge$ by

$$\ell_\wedge(q) = \bigwedge \mathsf{Con}(q)$$

Here, the operation $\wedge$ is the greatest lower bound operation in the non–structural lattice Σ. Notice that $\bigwedge \emptyset = \top$.

Given a constraint set C and a term τ, we define the tree $t^C_\wedge(\tau)$ as follows:

$$t^C_\wedge(\tau) = \lambda w.\ell_\wedge(\widehat{\delta}_D(\Uparrow_C(\tau), w))$$

Thus, the automaton $\mathcal{A}^C_D$ induces a regular tree $t^C_\wedge(\tau)$ for each term τ.

Similar definitions can be given for structural subtyping, once we have defined deterministic constraint automata for the structural theory. This will be done later.

9.1.6 Satisfiability and consistency

Using the methods of [57, 59], one can show the following theorem, which states that satisfiability coincides with consistency for non–structural subtyping (see also Section 2.4.3 for background). The theorem is an elegant use of constraint– and term automata. It shows that, whenever C is consistent, then the deterministic automaton $\mathcal{A}^C_D$ effectively translates C into a solution to itself through its induced term automata. Moreover, since $t^C_\wedge(\alpha)$ is a regular tree (as induced by a finite state automaton), we see that it is immaterial, from the perspective of the satisfiability problem, whether we work with general trees or just regular trees.

Definition 9.1.4 A constraint set is called *monotone*, if and only if it does not mention the constructor $\to$. □

Theorem 9.1.5 *Let C be a non–structural set. If C closed and consistent, then C has a regular solution given by the valuation $v_\wedge$ with the definition*

$$v_\wedge(\alpha) = \begin{cases} t^C_\wedge(\alpha) & \text{for } \alpha \in \mathsf{Var}(C) \\ \top & \text{for } \alpha \notin \mathsf{Var}(C) \end{cases}$$

If C is monotone, then this is the largest solution to C.

In particular, any constraint set is satisfiable if and only if it is consistent.

Theorem 9.1.5 leads to a PTIME (in fact, cubic time) decision procedure for satisfiability with non–structural (recursive) subtyping, because satisfiability reduces (by the theorem) to checking consistency, which, in turn, can be done by computing the closure of the constraint set; the set $\mathsf{Cl}(C)$ can be obtained from C in cubic time, using a dynamic transitive closure algorithm.

9.2 Prefix closed automata

In this section, we begin our proof that the entailment problem is PSPACE–hard. The proof is essentially a reduction from the universality problem for NFA's. This problem, called UNIV in the sequel, is:

- Given an NFA A over a non-trivial alphabet Σ, does A accept all strings, *i.e.*, is it the case that $L(A) = \Sigma^*$?

Here, $L(A)$ is the language accepted by A; an alphabet is non-trivial, if it contains more than one element. The problem UNIV is known to be PSPACE-complete [30].

However, we cannot reduce UNIV directly to entailment; rather, we shall need to reduce a *special* case of this problem to entailment. The reason is that a basic idea (found in both [65] and [24] but used differently here) of our reduction is to represent sets of strings as domains of trees, and since the latter are always *prefix-closed* languages, we shall need to consider only automata that accept such languages. We therefore begin, in this section, by proving that the universality problem for automata that accept only prefix-closed languages remains PSPACE-complete; we then give the reduction to entailment in Section 9.3.

In this section, Σ denotes an arbitrary, non-trivial alphabet (*i.e.*, Σ has more than one element.)

An NFA (non-deterministic finite automaton) is called *prefix closed*, if all its states are accept states. Thus, the only way a prefix closed NFA can fail to accept a word w is by getting into a stuck state on input w. Note that any language recognized by a prefix closed NFA must be a prefix closed language (a language L is prefix closed if every prefix of any string in L is again in L). Conversely, it is not difficult to see that every prefix–closed regular language is recognized by a prefix–closed NFA.

Let CLOSED-UNIV be the computational problem:

- Given a *prefix–closed* NFA A over a non-trivial alphabet Σ, is it the case that $L(A) = \Sigma^*$?

The construction used in the proof of the following lemma is due to Vaughan Pratt [61].

Lemma 9.2.1 *The problem CLOSED-UNIV is PSPACE-complete.*

PROOF Membership in PSPACE is obvious, since we know that UNIV is in PSPACE. To see hardness, we reduce from UNIV, as follows. Given an NFA A over Σ, to decide $L(A) = \Sigma^*$ we construct a prefix closed automaton $\overline{A}$ over an enlarged alphabet $\overline{\Sigma}$, as follows:

1. Add a new letter, $\overline{\sigma}$, to the alphabet Σ, and call the resulting enlarged alphabet $\overline{\Sigma}$.

2. Add a new state $\overline{q}$ with transitions on all letters out of it (including $\overline{\sigma}$); all transitions loop straight back to $\overline{q}$. (So starting in state $\overline{q}$, every string over $\overline{\Sigma}$ leads to that state.)

3. For each accepting state q' of A, add a $\overline{\sigma}$-transition from q' to the new state $\overline{q}$.

4. Make all states of the resulting automaton accepting, and call that automaton $\overline{A}$.

We will show that $L(A) = \Sigma^*$ if and only if $L(\overline{A}) = \overline{\Sigma}^*$.

Assume first, for the implication from right to left, that we have $L(\overline{A}) = \overline{\Sigma}^*$. Let w be an arbitrary word in Σ^*. Since $L(\overline{A}) = \overline{\Sigma}^*$, we have $w\overline{\sigma} \in L(\overline{A})$; inspection of $\overline{A}$ shows that this implies the existence in $\overline{A}$ of a w-transition from the start-state to some state which is an accept-state of A. But then there must already be a w-transition in A from the start-state of A to an accept-state of A, so $w \in L(A)$.

Assume now, for the implication from left to right, that we have $L(A) = \Sigma^*$, and let $\overline{w}$ be an arbitrary word in $\overline{\Sigma}$. If $\overline{w}$ is in Σ^*, we clearly have $\overline{w} \in L(\overline{A})$, since $\overline{A}$ certainly accepts everything accepted by A. We can therefore assume from now on that $\overline{w} \notin \Sigma^*$. Then we can write $\overline{w}$ as $\overline{w} = w\overline{\sigma}w'$ with $w \in \Sigma^*$ and $w' \in \overline{\Sigma}^*$. Since $L(A) = \Sigma^*$, there is a w-transition in A from the start-state to an accepting state q', and hence there is a w-transition in $\overline{A}$ from the start-state to q', and hence there is a $w\overline{\sigma}$-transition in $\overline{A}$ from the start-state to the state $\overline{q}$; since everything in $\overline{\Sigma}^*$ is accepted from $\overline{q}$, it follows that $\overline{A}$ accepts $\overline{w}$.

The above construction is a log-space reduction of UNIV to CLOSED-UNIV. □

Strictly speaking, the above reduction shows that the universality problem for closed automata over an alphabet with at least *three* letters is PSPACE-complete. However, it is easy to remove this limitation by standard techniques (we can encode a large alphabet into bit strings.)

9.3 PSPACE–hardness for recursive subtyping

Consider the two letter alphabet $\{f, s\} \subseteq \mathbf{A}$ and let ϵ denote the empty string. We will show that, for any prefix closed NFA A over $\{f, s\}$, we can construct a constraint set C_A such that $L(A) = \{f, s\}^*$ if and only if $C_A \models \alpha \leq \beta$, for some variables α and β in C_A. The types in C_A will be built from variables and the pairing constructor only. The resulting construction will be a log-space reduction of CLOSED-UNIV to the entailment problem $C \models \alpha \leq \beta$, thereby proving the latter problem PSPACE–hard, by Lemma 9.2.1.

We proceed to describe the construction of C_A. Assume we have given the prefix closed NFA $A = (Q, \{f, s\}, \delta, q_0, Q)$ (state set is Q, transition relation is δ, start-state is q_0, and – since the automaton is prefix closed – the set of accept states is the entire set of states, Q.) We write $q_i \mapsto^w q_j$ to indicate that there is a w-transition from state q_i to state q_j in A. If $w \in \{f, s\} \cup \{\epsilon\}$, we say that a transition $q_i \mapsto^w q_j$ is *simple*. A is completely determined by its simple transitions, since these just define its transition relation.

Suppose that A has n simple transitions, and assume them to be ordered in some arbitrary, fixed sequence. For each k'th simple transition $q_i \mapsto^w q_j$ we define a constraint set C_k; we associate a distinct variable α_i to each state q_i (so, in particular, the variable α_0 corresponds to the start state of A), we associate a distinct variable δ_k to each k, $1 \leq k \leq n$, and we further use two distinct variables, γ_k and γ'_k. The construction of C_k is as follows:

- If the k'th simple transition in A is $q_i \mapsto^f q_j$, then
$$C_k = \{\alpha_i \leq \alpha_j \times \delta_k\}$$
- If the k'th simple transition in A is $q_i \mapsto^s q_j$, then
$$C_k = \{\alpha_i \leq \delta_k \times \alpha_j\}$$
- If the k'th simple transition in A is $q_i \mapsto^\epsilon q_j$, then
$$C_k = \{\alpha_i \leq \alpha_j\}$$

Now define C_A to be

$$C_A = (\bigcup_{k=1}^{n} C_k) \cup \{\beta = \beta \times \beta\}$$

Notice that any solution to C_A must map β to the complete, infinite binary tree $t^\infty = \mu\gamma.\gamma \times \gamma$, hence β is just a name for that tree.

The constraint sets C_A have a particular form. In order to capture this technically, we give the following definition.

Definition 9.3.1 A simple and monotone constraint set C is called *directed* if and only if every inequality in C has either one of the following forms:

- $\alpha \leq \alpha'$
- $\alpha \leq \alpha_1 \times \alpha_2$

□

It is easy to see that all the sets C_A are directed, when the equation defining β is removed. Notice also that a directed set is trivially *closed under decomposition*, *i.e.*, for C directed we have $C^* = \mathsf{Cl}(C)$.

Entailment with directed sets is never completely trivial:

Lemma 9.3.2 *Every directed set has a solution.*

PROOF It is obvious from the form of directed sets that mapping every variable to $\bot$ results in a solution. □

We can now prove the main lemma for PSPACE–hardness.

Lemma 9.3.3 *Let A be a prefix closed NFA over $\{f, s\}$. Then*

$$L(A) = \{f, s\}^* \text{ if and only if } C_A \models \alpha_0 \leq \beta$$

PROOF First notice that, by construction of C_A, there is a transition $q_i \mapsto^w q_j$ in A if and only if there is a transition $\alpha_i \mapsto^w \alpha_j$ in the constraint automaton $\mathcal{A}_N^{C_A}$. Moreover, C_A is closed under decomposition, in the sense that one has $\mathsf{Cl}(C_A) = C_A^*$, because there are no upper bounds on constructed types in C_A (and hence the decomposition rule for closure cannot be used.) Because C_A is closed under decomposition and C_A is monotone, the set $\widehat{\delta}_D(\Uparrow_C(\alpha), w)$ is determined by the set of vertices reachable in $\mathcal{A}_N^{C_A}$ on a w–transition; here, δ_D is the transition function of $\mathcal{A}_D^{C_A}$.

We first prove the implication

$$L(A) \neq \{f, s\}^* \Rightarrow C_A \not\models \alpha_0 \leq \beta$$

So assume that there exists a word $w \in \{f, s\}^*$ such that $w \notin L(A)$. Since we can assume w.l.o.g. that A has at least one state, and since A is prefix closed, we have $\epsilon \in L(A)$, so $L(A) \neq \emptyset$. Then there is a prefix w' of w of maximal length such that $w' \in L(A)$; we can assume that w can be written as $w = w'fw''$ (the alternative case $w = w'sw''$ is similar.) Now, *because A is a prefix closed automaton*, it follows that for *any* state q_k such that $q_0 \mapsto^{w'} q_k$, there can be no state q_l such that $q_k \mapsto^f q_l$; inspection of the construction of C_A then shows that, for any k such that $q_0 \mapsto^{w'} q_k$, the only non-variable upper bounds in the transitive closure of C_A on α_k are of the form $\alpha_k \leq \delta_m \times \alpha_s$, where δ_m is unbounded in C_A; it follows that one has either $\widehat{\delta}_D(\alpha_0, w') = \emptyset$ or else $\widehat{\delta}_D(\alpha_0, w'f) = \emptyset$, and hence either $v_\wedge(\alpha_0)(w') = \top$ or else $v_\wedge(\alpha_0)(w'f) = \top$, where $v_\wedge$ is the largest solution to C_A, by Theorem 9.1.5. In either case, we have $v_\wedge(\alpha_0) \not\leq \beta$, thereby showing $C_A \not\models \alpha_0 \leq \beta$.

To prove the implication

$$L(A) = \{f, s\}^* \Rightarrow C_A \models \alpha_0 \leq \beta$$

let w be an arbitrary element in $\{f, s\}^*$. Then $wa \in L(A)$, $a \in \{f, s\}$, assuming the left hand side of the implication. Then there is a transition $q_0 \mapsto^w q_j \mapsto^a q_k$ for some q_j, q_k; by construction of C_A, there is a transition $\alpha_0 \mapsto^{wf} \alpha_k$ or $\alpha_0 \mapsto^{ws} \alpha_k$ in $\mathcal{A}_N^{C_A}$. Then, using that reachability in $\mathcal{A}_N^{C_A}$ determines $\widehat{\delta}_D$, and the simple form of C_A (only $\times$ and variables occur there), it is easy to verify that $w \in \mathcal{D}(v_\wedge(\alpha_0))$ with $\times \in \widehat{\delta}_D(\Uparrow(\alpha_0), w)$, and hence $v_\wedge(\alpha_0)(w) = \times$, where $v_\wedge$ is the largest solution to C_A, by Theorem 9.1.5. It follows that, for any $w \in \{f, s\}^*$, one has $v(\alpha_0)(w) \leq \times$, whenever v is a solution to C_A, thereby showing $C_A \models \alpha_0 \leq \beta$.

□

Theorem 9.3.4 *The problem of non-structural recursive subtype entailment is PSPACE–hard.*

PROOF By reduction from CLOSED-UNIV, using Lemma 9.3.3 together with Lemma 9.2.1. □

9.4 PSPACE–hardness for finite subtyping

In this section we show that the entailment problem remains PSPACE–hard, when considered over finite, non–structural trees, $\mathcal{T}_\Sigma^{\mathrm{F}}[n]$.

Anticipating Chapter 11, it will be shown that subtype entailment for structural subtyping over a lattice is coNP-complete. As we will see, membership in coNP follows from the fact that the non-entailment problem $C \not\models \alpha \leq \beta$ has a succinct certificate in the form of a word $w \in \mathbf{A}^*$ of length at most n (the size of the constraint set); intuitively, the certificate w identifies a common leaf address in $v(\alpha)$ and $v(\beta)$ for a possible solution v to C such that $v(\alpha)(w) \not\leq v(\beta)(w)$.

At first sight, it might be thought that a similar approach would be possible for non-structural subtyping over finite trees. However, it turns out that entailment here is PSPACE–hard. The reason, as we will see, is that infinite trees can be approximated in the non-structurally ordered space of finite trees, and one can use this to encode the universality problem for NFA's with entailment, just as was done for infinite trees.

When taken together with the coNP-completeness result of Chapter 11, our result shows that, unless NP = PSPACE, entailment with non-structural subtyping is strictly more difficult than entailment with structural subtyping, in the finite case.

The definition of entailment over the structure of finite trees $\mathcal{T}_\Sigma^{\mathrm{F}}$ is obtained by restricting valuations. Thus, for $v : \mathcal{V} \to \mathcal{T}_\Sigma^{\mathrm{F}}$, the satisfaction relation, written $v \models_{\mathrm{F}} \tau \leq \tau'$, holds if and only if $v(\tau) \leq v(\tau')$ holds in $\mathcal{T}_\Sigma^{\mathrm{F}}$, and entailment is defined by

$$C \models_{\mathrm{F}} \tau \leq \tau' \text{ if and only if } \forall v : \mathcal{V} \to \mathcal{T}_\Sigma^{\mathrm{F}}.\ v \models C \Rightarrow v \models \tau \leq \tau'$$

Since the order on $\mathcal{T}_\Sigma$ is a conservative extension of the order on $\mathcal{T}_\Sigma^{\mathrm{F}}$, satisfaction with valuations in $\mathcal{T}_\Sigma^{\mathrm{F}}$ is unchanged, *i.e.*, for $v : \mathcal{V} \to \mathcal{T}_\Sigma^{\mathrm{F}}$ one has $v \models_{\mathrm{F}} C$ if and only if $v \models C$.

In order to prove PSPACE–hardness of entailment with non-structurally ordered finite trees, we shall need to talk about approximations of infinite trees by finite trees. To this end, we recall from [6, 45] the definition of the *level-k truncation* of a tree t, denoted $t\lceil_k$; it has domain

$$\mathcal{D}(t\lceil_k) = \{w \in \mathcal{D}(t) \mid |w| \leq k\}$$

and is defined by

$$t\lceil_k(w) = \begin{cases} t(w) & \text{if } |w| < k \\ \bot & \text{if } |w| = k \end{cases}$$

This definition is simplified, since it does not take into account the contravariance of $\to$; this is so, because we shall only consider monotone constraint sets here.

We shall use the properties stated in the following lemma; the first two are taken from [6, 45], and the third is an immediate consequence of the definitions, so we leave out the proof.

Lemma 9.4.1 *Let t and t' be trees in $\mathcal{T}_\Sigma$. Then*

1. *$t \leq t'$ if and only if $\forall k \geq 0.\ t\lceil_k \leq t'\lceil_k$*
2. *$t\lceil_k \leq t\lceil_{k+1}$*
3. *$(t \times t')\lceil_{k+1} = t\lceil_k \times t'\lceil_k$*

As was exploited in the proof of Lemma 9.3.3, the constraint sets used to encode automata have a particularly simple form. The following lemma exploits this. In order to state the lemma, we need one more definition. If v is a valuation in $\mathcal{T}_\Sigma$, we define for all $k \geq 0$ the valuation $v\lceil_k$ in $\mathcal{T}_\Sigma^{\mathrm{F}}$ by

$$v\lceil_k(\alpha) = v(\alpha)\lceil_k$$

A key observation for proving PSPACE–hardness for finite trees is now the following lemma, which establishes that directed constraint sets over $\mathcal{T}_\Sigma[n]$ (as are used to simulate NFA's) can be solved with arbitrarily good finite approximations over $\mathcal{T}_\Sigma^{\mathrm{F}}[n]$. We let $\mathcal{T}_\Sigma$ be $\mathcal{T}_\Sigma[n]$ and let $\mathcal{T}_\Sigma^{\mathrm{F}}$ be $\mathcal{T}_\Sigma^{\mathrm{F}}[n]$.

Lemma 9.4.2 *Let C be directed, and let v be a valuation in $\mathcal{T}_\Sigma$ such that $v \models C$. Then one has*

$$\forall k \geq 1.\ v\lceil_k \models_{\mathrm{F}} C$$

in $\mathcal{T}_\Sigma^{\mathrm{F}}$.

PROOF All inequalities in C have one of the forms $\alpha \leq \alpha'$ or $\alpha \leq \alpha_1 \times \alpha_2$. It is sufficient to show that $v\lceil_k$ satisfies all such inequalities in $\mathcal{T}_\Sigma$, because the order on $\mathcal{T}_\Sigma$ is a conservative extension of the order on $\mathcal{T}_\Sigma^{\mathrm{F}}$:

- $\alpha \leq \alpha' \in C$: We have

$$\begin{array}{lll} v(\alpha) \leq v(\alpha') & \Rightarrow & (1) \\ v(\alpha)\lceil_k \leq v(\alpha')\lceil_k & \Leftrightarrow & \\ v\lceil_k(\alpha) \leq v\lceil_k(\alpha') & & \end{array}$$

- $\alpha \leq \alpha_1 \times \alpha_2 \in C$: We have

$$\begin{array}{lll} v(\alpha) \leq v(\alpha_1 \times \alpha_2) & \Rightarrow & (1) \\ v(\alpha)\lceil_k \leq v(\alpha_1 \times \alpha_2)\lceil_k & \Leftrightarrow & (3) \\ v(\alpha)\lceil_k \leq v(\alpha)\lceil_{k-1} \times v(\alpha')\lceil_{k-1} & \Rightarrow & (2) \\ v(\alpha)\lceil_k \leq v(\alpha)\lceil_k \times v(\alpha')\lceil_k & \Leftrightarrow & \\ v\lceil_k(\alpha) \leq v\lceil_k(\alpha) \times v\lceil_k(\alpha') & \Leftrightarrow & \\ v\lceil_k(\alpha) \leq v\lceil_k(\alpha_1 \times \alpha_2) & & \end{array}$$

The numbers refer to the properties of Lemma 9.4.1 justifying the implications. □

We can now show that entailment over $\mathcal{T}_\Sigma^{\mathrm{F}}$ is PSPACE–hard.

Given an NFA A, we define a constraint set C_A^{F}; it is defined exactly as C_A in Section 9.3, except that, instead of the equation $\beta = \beta \times \beta$, we now take the single inequality

$$\beta \times \beta \leq \beta \tag{9.1}$$

Lemma 9.4.3 *Let A be a prefix closed NFA over $\{f, s\}$. Then*

$$L(A) = \{f, s\}^* \text{ if and only if } C_A^{\mathrm{F}} \models_{\mathrm{F}} \alpha_0 \leq \beta$$

PROOF For $n \geq 0$, define the trees $t_n \in T_\Sigma^{\mathrm{F}}$ by

$$\begin{array}{lcl} t_0 & = & \top \\ t_{n+1} & = & t_n \times t_n \end{array}$$

Let t^∞ denote the complete, infinite binary tree $\mu\gamma.\gamma \times \gamma$. We have $t_n \times t_n \leq t_n$, $n \geq 0$, in $\mathcal{T}_\Sigma$, and the trees t_n can be regarded as increasingly good finite approximations "from above" of t^∞. Moreover, we clearly have $t^\infty \leq v(\beta)$ in $\mathcal{T}_\Sigma$ for any valuation v satisfying $\beta \times \beta \leq \beta$. Because C_A^{F} contains (9.1) and β occurs nowhere else in C_A^{F}, the possible solutions for β in C_A^{F} include all the trees $\{t_n \mid n \geq 0\}$, and any such solution for β can be combined with any solution for the other variables in C_A^{F} to a solution for the entire set C_A^{F}.

To prove the implication

$$L(A) = \{f, s\}^* \Rightarrow C_A^{\mathrm{F}} \models_{\mathrm{F}} \alpha_0 \leq \beta$$

assume $L(A) = \{f, s\}^*$, and let v be a valuation in $\mathcal{T}_\Sigma^{\mathrm{F}}$ satisfying C_A^{F}. The valuation v can be regarded as a valuation in $\mathcal{T}_\Sigma$, and it satisfies all the

inequalities in C_A except the ones for β in C_A. By the same argument as was used in the proof of Lemma 9.3.3, we have $v(\alpha_0) \leq t^\infty$ in the structure $\mathcal{T}_\Sigma$. Since $t^\infty \leq v(\beta)$, we have $v(\alpha_0) \leq v(\beta)$ in $\mathcal{T}_\Sigma$, and since these are finite trees, it also holds in $\mathcal{T}_\Sigma^{\mathrm{F}}$; this proves the implication.

To prove the implication

$$L(A) \neq \{f, s\}^* \Rightarrow C_A^{\mathrm{F}} \not\models_{\mathrm{F}} \alpha_0 \leq \beta$$

assume $L(A) \neq \{f, s\}^*$. By the argument used in the proof of Lemma 9.3.3, there is a valuation v in $\mathcal{T}_\Sigma$ such that $v \models C_A$ with $v(\alpha_0)(w) = \top$ for some w so that $v(\alpha_0) \not\leq t^\infty$. Let $C_A^- = C_A^{\mathrm{F}} \setminus \{\beta \times \beta \leq \beta\}$. Then C_A^- is clearly a directed set, and by Lemma 9.4.2 we have

$$\forall k \geq 0.\ v\lceil_k \models_{\mathrm{F}} C_A \tag{9.2}$$

in $\mathcal{T}_\Sigma^{\mathrm{F}}$. Let $k = |w| + 1$, then $v\lceil_k \models_{\mathrm{F}} C_A^-$, by (9.2), and we have $w \in \mathcal{D}(v\lceil_k(\alpha_0))$ with

$$v\lceil_k(\alpha_0)(w) = \top$$

Now, $v\lceil_k$ can be extended to a satisfying valuation v' of C_A^{F} by mapping β to t_n for any $n \geq 0$, because C_A^- is independent of β. By choosing n sufficiently large ($n = k$ will do) we get a valuation v' satisfying C_A^{F} and with $v'(\alpha_0)(w) = \top$ and $v'(\beta)(w) = \times$, thereby proving $C_A^{\mathrm{F}} \not\models_{\mathrm{F}} \alpha_0 \leq \beta$. □

Theorem 9.4.4 *The problem of non-structural subtype entailment over finite trees is PSPACE–hard.*

PROOF By reduction from CLOSED-UNIV, using Lemma 9.4.3 together with Lemma 9.2.1. □

We leave the upper bound problem open. In Chapter 12 we will give a PSPACE upper bound for structural recursive subtype entailment (and a matching lower bound). We believe that rather similar techniques can be employed to show that non–structural entailment is in PSPACE. However, we have not yet obtained a satisfactory correctness proof, and we confine ourselves to stating the following

Conjecture 9.4.5 *The problem of non–structural subtype entailment is in PSPACE and hence PSPACE–complete, both over finite and infinite trees.* □

Chapter 10

Structural trees

The purpose of this chapter is to develop concepts and methods which will be useful later, when we consider entailment over structural trees. As promised in Section 2.4.2, we show that satisfiability in infinite structural trees over a lattice is in PTIME (cubic time), thereby completing the classification of satisfiability complexity for lattice based structures (see the table in Section 2.4.4).

Recall from Section 2.1.4 that, in structural subtyping over a lattice, we have $\Sigma = L \cup \{\times, \rightarrow\}$ where $(L, \leq_L)$ is a lattice of constants, and Σ is ordered by taking the disjoint union of L, $\{\times\}$ and $\{\rightarrow\}$, *i.e.*, for $\sigma, \sigma' \in \Sigma$ we have $\sigma \leq_\Sigma \sigma'$ if and only if either σ and σ' are both in L and $\sigma \leq_L \sigma'$, or σ and σ' are both $\times$, or σ and σ' are both $\rightarrow$. In structural subtyping, only trees with the same "shape" (domain) can be compared, and comparability is reducible to the order $\leq_L$ on constants at the leaves. This was captured technically in Lemma 2.1.1. We gave a number of further standard properties for finite, structural subtyping in Chapter 2. These will be generalized to the recursive case in the present chapter. We fix $\mathcal{T}_\Sigma$ to be $\mathcal{T}_\Sigma[s]$ in this chapter.

As was noted in Section 2.4, previous work by Palsberg and O'Keefe [57] and by Pottier [59] has shown that satisfiability of constraint sets in recursive, non–structural subtyping is equivalent to consistency and therefore in PTIME (recall Theorem 2.4.3). The method used to prove this was to consider deterministic constraint automata, which translate a constraint set to a solution, whenever the constraint set is consistent. See Section 9.1.6 with Theorem 9.1.5. As was also noted earlier, in Section 2.4, no exact classification has so far been given for the *structural*, recursive case. The best previous result is the PSPACE lower bound and DEXPTIME upper

bound for the problem REG–SAT (satisfiability of infinite, regular sets of atomic inequalities) of Tiuryn and Wand [72]. We prove here that structural recursive subtype satisfiability is equivalent to consistency, when the order is generated from a lattice of base types, and therefore the satisfiability problem is in PTIME for this model also.

In the light of the previous results for the non–structural case by Palsberg–O'Keefe and Pottier, it may not be so surprising that a similar approach would work for structural, recursive subtyping, and, indeed, our method is an adaptation of theirs. Nevertheless, adapting the method in detail to the structural model requires a good amount of technical work, which (even though it may be found to be somewhat boring) requires some care. A sketch of an alternative reduction of satisfiability to consistency can be found in the note [26] by Frey. [1]

10.1 Infinitary matching

Infinite terms

In order to talk about matching properties of constraint sets, it is useful to introduce special terms and substitutions. We first introduce a form of infinitary type expressions. A *generalized term* is a tree in $\mathcal{T}_{\Sigma\cup\mathcal{V}}$, where variables in $\mathcal{V}$ are regarded as constructors of arity 0 (however, only non-variable constructors of arity 0 are called constants); thus, a generalized term is just like a term, but with possibly infinite domain. We write $\mathcal{T}_\Sigma^\infty(\mathcal{V})$ to denote the set of generalized terms. Valuations (mappings from $\mathcal{V}$ to $\mathcal{T}_\Sigma$) and substitutions (mappings from $\mathcal{V}$ to $\mathcal{T}_\Sigma^\infty(V)$) can be considered as homomorphisms on generalized terms in the obvious way. We can also consider *generalized constraint sets*, which are finite sets of inequalities of the form $\theta \leq \theta'$, where θ and θ' are generalized terms. The notion of matching constraint sets carries over in the obvious way.

Lemma 10.1.1 *(Leaf Lemma) Let θ_1, θ_2 be generalized terms with $\mathcal{D}(\theta_1) = \mathcal{D}(\theta_2)$, and let v be a valuation, $v : \mathcal{V} \to T_\Sigma$. Then $v(\theta_1) \leq v(\theta_2)$ holds in T_Σ if and only if $v(\theta_1(w)) \leq^w v(\theta_2(w))$ for every (common) leaf address w in θ_1 and θ_2.*

PROOF The lemma is an easy consequence of Lemma 2.1.1 together with the fact that a valuation preserves constructors homomorphically. □

[1] I am grateful to Alexandre Frey for illuminating discussions about these matters.

Most general matching substitutions

Let U_C be the most general (possibly circular) unifier of E_C for a weakly unifiable set C (see Definition 2.3.1). If C is a structural set, let $u_\alpha^C = U_C(\alpha)$, let $\{\alpha_w \mid w \in \mathbf{A}^*\}$ be a set of variables not occurring in C. For each variable α in C define the generalized term θ_α^C with $\mathcal{D}(\theta_\alpha^C) = \mathcal{D}(u_\alpha^C)$ and with

$$\theta_\alpha^C(w) = \begin{cases} u_\alpha^C(w) & \text{if } w \in \mathsf{In}(u_\alpha^C) \\ \alpha_w & \text{if } w \in \mathsf{Lf}(u_\alpha^C) \end{cases}$$

Define the substitution Θ_C from terms to generalized terms by setting

$$\Theta_C(\alpha) = \theta_\alpha^C$$

A *matching substitution* for a structural, recursive constraint set C is a substitution $S : \mathcal{V} \to T_\Sigma^\infty(\mathcal{V})$ such that $S(C)$ is matching. If C is weakly unifiable, then it has a matching substitution, and Θ_C is a *most general matching substitution* for C, *i.e.*, $\Theta_C(C)$ is matching, and for any other matching substitution S for C there is a substitution R such that $S = R \circ \Theta_C$ holds on the variables occurring in C. This is a straight-forward generalization of the corresponding property from finite terms (Section 2.3.1, see [53] for the finite case, and see [72, 27] for other generalizations to the infinitary case.)

Next we give a simple example involving infinite terms introduced by Θ_C, arising due to circular constraints.

Example 10.1.2 (Infinite expansion)
Let

$$C = \{\alpha \leq \alpha \to \beta\}$$

Then $\Theta_C(C)$ produces the inequality with infinite terms shown in Figure 10.1. The domain of the type $\Theta_C(\alpha)$ is found by unifying E_C. □

The Match Lemma holds in the case of structural recursive subtyping, with the same proof as that of Lemma 2.3.5, only now using the generalized notion of terms:

Lemma 10.1.3 *(Match Lemma)*

1. *If $v \models C$, then $v(C)$ is matching.*
2. *If $v \models C$, then $v = v' \circ \Theta_C$ holds on the variables in C for some valuation v'*
3. *$C \models \tau \leq \tau'$ if and only if $\Theta_C(C) \models \Theta_C(\tau) \leq \Theta_C(\tau')$*

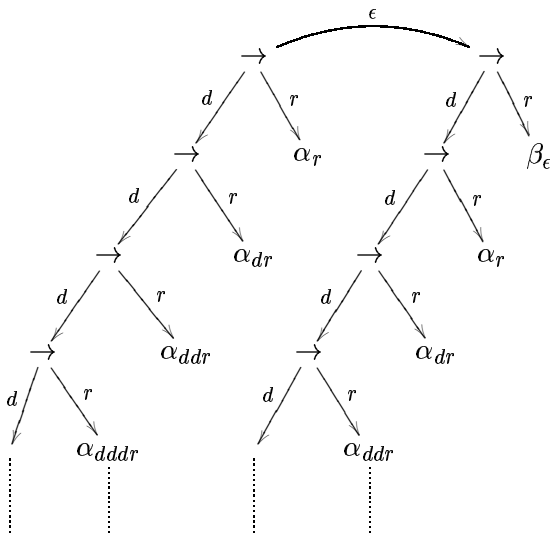


Figure 10.1: Constraint set $C = \{\alpha \leq \alpha \to \beta\}$ expanded by Θ_C

Flattening

For a weakly unifiable set C, define the *flattened* set $C^\flat$ by

$$C^\flat = \{\theta(w) \leq^w \theta'(w) \mid \theta \leq \theta' \in \Theta_C(C), w \in \mathsf{Lf}(\theta) \cap \mathsf{Lf}(\theta')\}$$

Observe that, if C is weakly unifiable, then $\mathsf{Lf}(\theta) = \mathsf{Lf}(\theta')$ for all $\theta \leq \theta' \in \Theta_C(C)$. Notice also that a generalized type $\Theta_C(\tau)$ may be a non-regular tree, because it could contain infinitely many distinct variables. Notice finally that the set $C^\flat$ may be infinite.

Given constraint set C and variables α, β we define the set of inequalities $[\alpha \leq \beta]_C$ to be the flattened set

$$[\alpha \leq \beta]_C = \{\theta^C_\alpha(w) \leq^w \theta^C_\beta(w) \mid w \in \mathsf{Lf}(\theta^C_\alpha) \cap \mathsf{Lf}(\theta^C_\beta)\}$$

Example 10.1.4 Consider again the set C from Example 10.1.2 and Figure 10.1. We have

$$\begin{array}{rcll} C^\flat & = & \{\alpha_r \leq \beta_\epsilon\} & \\ & \cup & \{\alpha_{d^n r} \leq \alpha_{d^{n+1} r}\} & n \geq 0, n \text{ even} \\ & \cup & \{\alpha_{d^{n+1} r} \leq \alpha_{d^n r}\} & n \geq 0, n \text{ odd} \end{array}$$

□

Recall from Definition 2.3.9 the notion of variables *matching in* a constraint set. This notion evidently transfers to the setting of infinite terms as well. The following lemma is a consequence of elementary properties of the most general matching substitution. It is a straight–forward generalization of Lemma 2.3.10 from the finite case; the wording of the lemma remains the same, and so does its proof. We repeat the lemma here for convenience:

Lemma 10.1.5 *(Flattening Lemma) Let C be a weakly unifiable constraint set. Then*

1. *C is satisfiable if and only if $C^\flat$ is satisfiable. More specifically, one has*
   (a) *If $v \models C$, then there is a valuation v' such that $v = v' \circ \Theta_C$ holds on the variables in C, and with $v' \models C^\flat$*
   (b) *$v \models C^\flat$ if and only if $v \circ \Theta_C \models C$*
2. *$v \models C^\flat$ if and only if $v \circ \Theta_C \models C$*
3. *If α and β are matching in C, then*
$$C \models \alpha \leq \beta \text{ if and only if } C^\flat \models [\alpha \leq \beta]_C$$

PROOF The proof is the same as the proof of Lemma 2.3.10, using the generalized definition of matching substitutions and terms. □

10.2 Representation

In this section we show that satisfiability and entailment over $\mathcal{T}_\Sigma$ can (under certain conditions) be faithfully represented in a substructure of $\mathcal{T}_\Sigma$. To this end, we introduce the notion of C-shaped valuations (a precise definition is given later.) Intuitively, a C-shaped valuation for a set C is one which maps every variable α in C to a tree with shape $\Theta_C(\alpha)$, *i.e.*, v uses a "minimal amount" of the structure $\mathcal{T}_\Sigma$ and contains no "garbage". It is the goal of this section to establish that, under certain conditions, satisfiability and entailment problems are completely determined by the behaviour of C-shaped valuations. This turns out to be a very useful property for defining constraint automata for constraint sets over structurally ordered trees.

Recall from Definition 8.3.2 that a variable α is said to be *equivalent to a constant* in an atomic constraint set C if and only if every valuation satisfying C maps α to a constant in L.

C–shaped valuations

Definition 10.2.1 We say that a valuation $v : \mathcal{V} \to \mathcal{T}_\Sigma$ is *C-shaped* if and only if $v(\alpha)$ matches $\Theta_C(\alpha)$ for every variable α occurring in C. Hence, if v is C-shaped, then v maps α to an element in a component lattice L_t where t matches $\Theta_C(\alpha)$, and if $v \models C$, then $v = v_L \circ \Theta_C$ for some $v_L : \mathcal{V} \to L$ (the equality holding on the variables in C). □

Non–trivially related variables

Definition 10.2.2 We say that α and β are *non-trivially related in C* if and only if the following conditions are satisfied:

- both α and β occur in C
- for all $A \leq A' \in [\alpha \leq \beta]_C$, either both A and A' are equivalent to a constant in $[\alpha \leq \beta]_C$, or else neither of them is
- α and β are matching in C

If one of these conditions is not satisfied, then we say that α and β are *trivially related* in C. □

The following theorem states that satisfiability and (under certain conditions) entailment over $\mathcal{T}_\Sigma$ can be faithfully represented in the structure

$$\mathcal{T}_C = \biguplus\{L_t \mid t \text{ matches } \Theta_C(\tau), \tau \text{ subterm in } C\}$$

that is, $\mathcal{T}_C$ is the least union of lattices L_t containing the range of C-shaped valuations applied to inequalities in C.

Theorem 10.2.3 *(Representation in $\mathcal{T}_C$) For any constraint set C one has:*

1. *C is satisfiable in $\mathcal{T}_\Sigma$ if and only if C is satisfied by a C-shaped valuation*
2. *If C is satisfiable and α, β are non-trivially related in C, then $C \models \alpha \leq \beta$ if and only if it holds for any C-*shaped *valuation v that $v \models \alpha \leq \beta$ whenever $v \models C$.*

PROOF See Appendix A.4 □

Lemma 10.2.4 *Let C be satisfiable. If α and β are trivially related in C, then $C \not\models \alpha \leq \beta$.*

PROOF See Appendix A.4 □

10.3 Structural constraint automata

We can modify the construction of constraint automata from Section 9.1 to work for structural subtyping. The non-deterministic automaton $\mathcal{A}_N^C$ can be defined as before. However, the deterministic version does not transfer, mainly because the labeling functions are not defined on empty sets, in the absence of a global top element in the structural ordering. As a consequence, the value

$$\ell_\wedge = \bigwedge \mathsf{Con}(q)$$

will be undefined in case $q = \emptyset$ (of course, the lattice operation above might be undefined in the case of structural subtyping for other reasons than $q = \emptyset$, but that is not so serious). This can occur for states q reachable in $\mathcal{A}_D^C$, whenever a set of variables $\{\alpha_i\}_i$ is reached with $q = \bigcup_i \Uparrow_C (\alpha_i) = \emptyset$ or $q = \bigcap_i \Uparrow_C (\alpha_i) = \emptyset$. One way of solving this problem, therefore, would be to make sure that, for all such sets $\{\alpha_i\}_i$, one would have $\bigcap_i \Uparrow_C (\alpha_i) \neq \emptyset$.

Using our results on representation of the previous section we can accomplish this by picking the top element of an appropriately chosen lattice L_t. The idea is, roughly, that every variable α in the constraint set C will be qualified with an inequality $\alpha \leq \mathrm{top}^{s(\alpha)}$, where $s(\alpha)$ is the shape of $\Theta_C(\alpha)$ and $\top^{s(\alpha)}$ is the top element of the lattice of trees of shape $s(\alpha)$. To make this precise, it is useful to introduce the notion of a shape. A *shape* is a tree in $\mathcal{T}_{\{\times,\to,\star\}}$. If C is a structural set, then we can assign a shape to each variable α in C, denoted $s_C(\alpha)$, which is the tree in $\mathcal{T}_{\{\times,\to,\star\}}$ matching $\Theta_C(\alpha)$. We call the function s_C the *shape map associated with* C. It has support $\mathsf{Var}(C)$, *i.e.*, it is a total map on $\mathcal{V}$ acting as the identity outside $\mathsf{Var}(C)$. Notice that this means that, for variables α, β not in $\mathsf{Var}(C)$, we have $s_C(\alpha) = s_C(\beta)$ if and only if $\alpha = \beta$. Notice that $s_C(\alpha)$ is always a regular tree (even though $\Theta_C(\alpha)$ may not be), since it arises from the unification graph underlying U_C by insertion of $\star$ at the leaves of $U_C(\alpha)$. The regular tree $s_C(\alpha)$ can be regarded as a term automaton recognizing the domain of $\Theta_C(\alpha)$. (Notice that the map s_C depends on a given weakly unifiable set C, but for notational simplicity we will not always make this explicit, when the set C is clear from context.)

The idea then is to use the lattice $L_{s(\alpha)}$, that is

$$L_{s(\alpha)} = \{t \in \mathcal{T}_\Sigma \mid t \text{ matches } \Theta_C(\alpha)\}$$

Let s be a shape. Define the regular tree top^s with domain $\mathcal{D}(s)$ and

given by

$$\mathrm{top}^s(w) = \begin{cases} \top & \text{if } w \in \mathsf{Lf}(s) \text{ and } \pi w = 0 \\ \bot & \text{if } w \in \mathsf{Lf}(s) \text{ and } \pi w = 1 \\ s(w) & \text{if } w \in \mathsf{In}(s) \end{cases}$$

It is easy to see that top^s is the top element in the component lattice L_s (prove $t(w) \leq^w \mathrm{top}^s(w)$ for all $t \in L_s$ by induction on $|w|$, $w \in \mathcal{D}(t)$).

We wish to extend a constraint set C with inequalities of the form $\alpha \leq \mathrm{top}^{s(\alpha)}$, for all $\alpha \in \mathsf{Var}(C)$, to obtain an extended set C', where the new inequalities force that C' can only be solved by C-shaped valuations. However, for some rather subtle technical reasons (that will become clear) we will do this in a particular way. First, we wish to keep the constraint set finite and *simple*, so we cannot mention the (possibly infinite) trees $\mathrm{top}^{s(\alpha)}$ directly in C'. However, as regular trees, the elements $\mathrm{top}^{s(\alpha)}$ can all be defined by sets of *regular and contractive equations* (see [16]), of the form $\gamma = \gamma_1 \times \gamma_2$ or $\gamma = \gamma_1 \to \gamma_2$, which are evidently expressible by *simple* inequalities. Such sets have unique solutions (they define contractions in the complete metric space of trees, see [16].)

We can assume, therefore, that for any structural constraint set C and any shape s, $\alpha \in \mathsf{Var}(C)$, there is a simple constraint set C^s of regular equations defining top^s. We assume that the defining equations in the C^s use variables, ranged over by γ which occur nowhere else, and we let Γ denote the set of those variables. We can further assume that C^s has a variable γ_s representing the top level definition of top^s, *i.e.*, γ_s is a variable in C^s such that the unique solution to C^s maps γ^s to top^s.

Notice that, for any γ in C^s, γ has a non-variable upper bound, *i.e.*, there is an inequality $\gamma \leq \tau$ in C^s where τ is a constructed type or a constant ($\bot$ or $\top$).

We now define an extension of a given set C, called $C^\top$. The set $C^\top$ is just another version of C' where all inequalities are simple and all variables γ involved in defining the elements top^s have been identified, whenever they have the same shape. The set $C^\top$ is defined in three steps, as follows:

(*i*) Let s_C be the shape map associated with C, and let S be the set of shapes of variables in C, *i.e.*, $S = \{s_C(\alpha) \mid \alpha \in \mathsf{Var}(C)\}$. Define C_Γ by

$$C_\Gamma = C \cup \{\alpha \leq \gamma_s \mid \alpha \in \mathsf{Var}(C), s = s_C(\alpha)\} \cup \bigcup_{s \in S} C^s$$

(*ii*) Let s_Γ be the shape map associated with C_Γ. For each shape s in $\{s_\Gamma(\gamma) \mid \gamma \in \Gamma\}$ we assume a variable δ_s occurring nowhere else, and

we let Δ denote that set of variables. Define C_Δ by

$$C_\Delta = C_\Gamma \cup \{\gamma = \delta_s \mid \gamma \in \Gamma, s = s_\Gamma(\gamma)\}$$

(*iii*) Define $C^\top$ by

$$C^\top = \mathsf{Cl}(C_\Delta)$$

Define now $\Uparrow_C(\alpha)$ to be the set of non–variable terms that are upper bounds in C for α:

$$\Uparrow_C(\alpha) = \{\tau \mid \alpha \leq \tau \in C\} \setminus \mathcal{V}$$

If C is a weakly unifiable set and T is a set of terms, then we say that T *is matching with respect to* C if and only if all terms in T have the same shape under the shape map of C, *i.e.*,

$$\forall \tau, \tau' \in T.\ s_C(\tau) = s_C(\tau')$$

Lemma 10.3.1 *Let C be weakly unifiable. If V is a non-empty subset of $\mathsf{Var}(C^\top)$ and V is matching with respect to $C^\top$, then*

$$\bigcap_{\alpha \in V} \Uparrow_{C^\top}(\alpha) \neq \emptyset$$

PROOF See Appendix A.4 □

Corollary 10.3.2 *For any weakly unifiable constraint set C one has:*

1. *C is satisfiable if and only if $C^\top$ is satisfiable*
2. *If C is satisfiable and α, β are non-trivially related in C, then $C \models \alpha \leq \beta$ if and only if $C^\top \models \alpha \leq \beta$.*

PROOF See Appendix A.4 □

We define the deterministic constraint automaton $\mathcal{A}_D^C$ as before, assuming that C is *simple* and *closed*. Only now we do not have transitions from a set of terms unless all terms have the same top level constructor:

- for $q = \{\alpha_i \to \alpha_i'\}_{i \in I}$

$$\delta_D(q, d) = \textstyle\bigcap_{i \in I} \Uparrow_C(\alpha_i)$$
$$\delta_D(q, r) = \textstyle\bigcup_{i \in I} \Uparrow_C(\alpha_i')$$

- For $q = \{\alpha_i \times \alpha_i'\}_{i \in I}$:

$$\delta_D(q, f) = \bigcup_{i \in I} \Uparrow_C (\alpha_i)$$
$$\delta_D(q, s) = \bigcup_{i \in I} \Uparrow_C (\alpha_i')$$

- For q not of the forms above, the transition functions are undefined.

We can now supply a labeling function $\ell_\wedge$. We define

$$\ell_\wedge(q) = \begin{cases} \times & \text{if } \mathsf{Con}(q) = \{\times\} \\ \to & \text{if } \mathsf{Con}(q) = \{\to\} \\ \bigwedge q & \text{if } q \subseteq L \end{cases}$$

The function $\ell_\wedge$ is partial, since it is only defined for non-empty sets q such that either all types in q have the same top level constructor or else q is a set of constants.

For a structural set C and $\alpha \in \mathsf{Var}(C)$, we can induce a term (automaton) $t_\wedge^C(\alpha)$ as we did in the non-structural case:

$$t_\wedge^C(\alpha) = \lambda w.\ell_\wedge(\widehat{\delta}_D(q_\alpha, w))$$

where

$$q_\alpha = \Uparrow_{C^\top} (\alpha)$$

and $\widehat{\delta}_D$ is now the transition function of $\mathcal{A}_D^{C^\top}(q_\alpha)$ (notice the use of $C^\top$ here rather than C.) Our definitions are meaningful:

Lemma 10.3.3 *If C is weakly unifiable and $\alpha \in \mathsf{Var}(C)$, then $t_\wedge^C(\alpha)$ is a well defined term automaton (i.e., its labeling function is defined on all states reachable from the start state of the automaton.)*

PROOF See Appendix A.4 □

10.4 Satisfiability and consistency

We will show that a structural recursive constraint set is satisfiable if and only if it is consistent. The proof is similar to Pottier's proof [59] of the corresponding property for non–structural recursive sets; this proof, in turn, was based on ideas in [57]. The essential property needed is the following:

Lemma 10.4.1 *(Main Lemma) Suppose that C is consistent (weakly unifiable and ground consistent). Let $\Uparrow$ denote the function $\Uparrow_{C^\top}$. Let $\tau_1 \leq \tau_2 \in C^\top$ and $q_1 = \Uparrow(\tau_1)$, $q_2 = \Uparrow(\tau_2)$. Then , for any string $w \in \mathbf{A}^*$ accepted by both $\mathcal{A}_D^{C^\top}(q_1)$ and $\mathcal{A}_D^{C^\top}(q_2)$ one has*

$$\ell_\wedge(\widehat{\delta}_D(q_1, w)) \leq^w \ell_\wedge(\widehat{\delta}_D(q_2, w))$$

PROOF See Appendix A.4 □

Theorem 10.4.2 *C is satisfiable in the structurally ordered set of trees $\mathcal{T}_\Sigma$ if and only if C is consistent (weakly unifiable and ground consistent).*

PROOF It is obvious that any satisfiable set must be weakly unifiable and ground consistent. To see the other implication, suppose that C is weakly unifiable and ground consistent. Define the valuation $v_\wedge$ by setting

$$v_\wedge(\alpha) = t_\wedge^C(\alpha)$$

For a type expression τ, let $A(\tau)$ be the term automaton $\mathcal{A}_D^C(\Uparrow_C(\tau))$, and let $t_{A(\tau)}$ denote the term defined by this automaton (in the standard way, according to [45]). Because $\Uparrow_C(\tau) = \{\tau\}$, whenever τ is not a variable, one has

$$v_\wedge(\tau) = t_{A(\tau)}$$

for non-variable τ. Lemma 10.4.1 therefore states that $v_\wedge(\tau) \leq v_\wedge(\tau')$ for all $\tau \leq \tau' \in C$. Hence, $v_\wedge$ is a solution to C. □

Since weak unifiability of a constraint set is decidable by cyclic unification in almost linear time, and ground consistency can be checked by computing constraint closure using a dynamic transitive closure algorithm, we have:

Corollary 10.4.3 *Satisfiability of a constraint set in the structurally ordered set of trees $\mathcal{T}_\Sigma$ can be decided in time $O(n^3)$ (where n is the size of the constraint set).*

Chapter 11

Structural finite entailment

In this chapter we study the complexity of deciding entailment over structurally ordered finite trees. We prove that the problem is coNP–complete, even in the presence of just a single covariant constructor ($\times$).

To recall, the finite structural entailment $C \models \alpha \leq \beta$ holds if and only if

$$\forall v : \mathcal{V} \to \mathcal{T}_\Sigma^{\mathrm{F}} \,.\, v \models C \Rightarrow v \models \alpha \leq \beta$$

where $\mathcal{T}_\Sigma^{\mathrm{F}}$ is short notation for the set of finite trees with the structural order, $\mathcal{T}_\Sigma^{\mathrm{F}}[s]$.

We may note immediately that there is a naive algorithm to decide this problem. The algorithm proceeds by first flattening the constraint set C, using Lemma 2.3.10), to obtain the equivalent, *atomic* entailment problem, $C^\flat \models [\alpha \leq \beta]_C$. This problem, in turn, can be solved in time linear in the size of $C^\flat$, using the characterization of atomic entailment developed in Chapter 8. However, flattening a set can induce an exponential blow–up in the size of the resulting set, *i.e.*, $C^\flat$ may be of size 2^n where n is the size of C, because a most general matching substitution may incur such a blow–up (recall Example 2.3.7). The algorithm sketched above would therefore only yield an exponential time upper bound.

11.1 Leaf entailment

A restriction of entailment, which makes sense for structural subtyping systems, is to ask for entailment *at a given address*. This amounts, for instance, to querying whether a certain component of a variable (known to denote a

product) has to be less than the corresponding component of β (also known to denote a product.)

In more detail, suppose that C is such that α and β have the same shape in C (otherwise, the entailment $C \models \alpha \leq \beta$ is uninteresting), *i.e.*, $\Theta_C(\alpha)$ and $\Theta_C(\beta)$ have the same domain, and let w be a leaf address in $\Theta_C(\alpha)$ and $\Theta_C(\beta)$. We then say that C *w–entails* $\alpha \leq \beta$, written $C \models^w \alpha \leq \beta$, if and only if

$$\forall v : \mathcal{V} \to \mathcal{T}_\Sigma.\ v \models C \Rightarrow v(\alpha)(w) \leq^w v(\beta)(w)$$

The notion of leaf entailment gives rise to the following decision problem:

- Given constraint set C and a leaf address w for $\Theta_C(\alpha)$, decide whether $C \models^w \alpha \leq \beta$.

The notion of leaf entailment may be of practical interest in its own right; however, in this thesis, we shall use it for theoretical purposes later. It is easy to see that leaf entailment characterizes entailment completely, in the case of structural subtyping:

Proposition 11.1.1 *Let C be consistent, and suppose that α and β are matching in C. Then $C \models \alpha \leq \beta$ if and only if $C \models^w \alpha \leq \beta$ for every leaf address w in $\Theta_C(\alpha)$ and $\Theta_C(\beta)$.*

PROOF The proposition is an easy consequence of Lemma 2.3.8 together with Lemma 10.1.3. □

It is not too difficult to see that one can decide leaf entailment in PTIME. For now we will restrict ourselves to entailment over *C–shaped valuations* only (recall Definition 10.2.1). As we shall see later, the generalization to arbitrary valuations is easy enough and remains within PTIME for leaf entailment.

We will begin by describing a conceptually simple (if inefficient) algorithm. For now, we will assume that C is satisfiable (otherwise entailment trivially holds), and that the given address w is a leaf address for both $\Theta_C(\alpha)$ and $\Theta_C(\beta)$. So assume we are given a satisfiable set C, a goal inequality $\alpha \leq \beta$ and an appropriate address w. We will show how to decide the complementary, non–entailment problem

$$C \not\models^w \alpha \leq \beta \tag{11.1}$$

over C–shaped valuations. In order to do so, we shall need the notion of a *w–template*:

Definition 11.1.2 For a word $w \in \mathbf{A}^*$, we call a finite term T^w a *w-template* if $\mathcal{D}(T^w)$ is the least tree domain containing w and the leaves contain fresh, pairwise distinct variables. □

Now, let T^w_α and T^w_β be two distinct w–templates, each having variables disjoint from each other and C. Let $\alpha_w = T^w_\alpha(w)$ and $\beta_w = T^w_\beta(w)$, and let $C' = C\{\alpha \mapsto T^w_\alpha, \beta \mapsto T^w_\beta\}$. It is then easy to see that an address w satisfies (11.1) over C–shaped valuations, if and only if there exist constants b_1 and b_2 in L such that

1. $b_1 \not\leq^w_L b_2$, and
2. $C'' = C'\{\alpha_w \mapsto b_1, \beta_w \mapsto b_2\}$ is satisfiable

Soundness and completeness of this test follows, because any C–shaped valuation v such that $v \models C$ and $v(\alpha) \not\leq^w_L v(\beta)$ can evidently be obtained by composition with the substitution which maps α to $T^w_\alpha\{\alpha_w \mapsto b_1\}$ and β to $T^w_\beta\{\beta_w \mapsto b_2\}$ for some $b_1, b_2 \in L$ with $b_1 \not\leq^w b_2$. These observations are summarized in the algorithm shown in Figure 11.1.

The algorithm runs in PTIME, because satisfiability is in PTIME, by Theorem 2.4.2. The algorithm presented in Figure 11.1 is rather naive and inefficient. It is similar to the naive algorithm for deciding atomic entailment by explicit representation of the negation of inequalities, via an exhaustive search for constants b_1, b_2 in L such that $b_1 \not\leq_L b_2$ combined with repeated checks for satisfiability (see Section 8.2).

We can do better if we take into account our linear time characterization of atomic entailment, contained in Theorem 8.1.2. Such a solution is shown in Figure 11.2, which calls a linear time entailment procedure for atomic constraints as a subroutine. It uses an operation At on constraint sets which we shall define next.

Say that a variable α occurring in C is *atomic in* C if and only if $U_C(\alpha)$ is a constant or a variable (*i.e.*, α does not get compared to a structured type under unification.) Such a variable can evidently be mapped to a constant in L under a satisfying valuation (if any such exists.) We say that a constant is trivially atomic in C. Then define the *atomic* constraint set $\mathsf{At}(C)$ by

$$\mathsf{At}(C) = \{A \leq A' \in C \mid A \text{ and } A' \text{ are atomic in } C\}$$

Correctness of the algorithm shown in Figure 11.2 can be seen as follows. If the atomic entailment test $\widehat{C} \models_L \alpha_w \leq^w \beta_w$ succeeds, the leaf entailment is easily seen to hold (details are left to the reader.) Conversely, if the atomic

Given C, α and β with C satisfiable and w a leaf address in both $\Theta_C(\alpha)$ and $\Theta_C(\beta)$. The algorithm accepts if and only if $C \models^w \alpha \leq^w \beta$ over C–shaped valuations in finite trees ordered structurally.

1. Build w–templates T^w_α and T^w_β;

2. Initialize for Loop:
   $\alpha_w := T^w_\alpha(w)$;
   $\beta_w := T^w_\beta(w)$;
   $C' := C\{\alpha \mapsto T^w_\alpha, \beta \mapsto T^w_\beta\}$;

3. Loop:

   for $(b_1, b_2) \in L \times L$ do
   if $b_1 \not\leq^w_L b_2$ then
   $C'' := C'\{\alpha_w \mapsto b_1, \beta_w \mapsto b_2\}$;
   if C'' is satisfiable then REJECT;

4. If step 3 did not lead to rejection, then ACCEPT.

Figure 11.1: PTIME algorithm for leaf–entailment.

entailment test fails, then there must exist constants $b_1, b_2 \in L$ with $b_1 \not\leq^w_L b_2$ such that $\widehat{C}\{\alpha_w \mapsto b_1, \beta_w \mapsto b_2\}$ is consistent (and hence satisfiable); then the set $C''\{\alpha_w \mapsto b_1, \beta_w \mapsto b_2\}$ must also be consistent (this should be rather obvious, but a detailed argument for this can be found in the proof of Theorem 5 in [71]), and it easily follows that C becomes consistent when α is mapped to $T^w_\alpha\{\alpha_w \mapsto b_1\}$ and β is mapped to $T^w_\beta\{\beta_w \mapsto b_2\}$, thereby constituting a counterexample to the leaf entailment.

By generalizing the characterization of atomic entailment found in Theorem 8.1.2, it is possible to come up with a more sophisticated algorithm, which avoids constructing the templates for α and β and computing closure explicitly. However, in order to avoid tedious repetitions, we postpone doing so until we have given a similar generalization suited for recursive structural subtyping (Chapter 12); this will contain the techniques necessary for an improved algorithm for both finite subtype entailment and leaf entailment as well. For now, we limit ourselves to recording

Proposition 11.1.3 *Assume that C is satisfiable and that w is a leaf ad-*

Given C, α and β with C satisfiable and w a leaf address in both $\Theta_C(\alpha)$ and $\Theta_C(\beta)$. The algorithm accepts if and only if $C \models^w \alpha \leq^w \beta$ over C–shaped valuations in finite trees ordered structurally.

1. Build w–templates T^w_α and T^w_β;

2. $\alpha_w := T^w_\alpha(w)$;
   $\beta_w := T^w_\beta(w)$;
   $C' := C\{\alpha \mapsto T^w_\alpha, \beta \mapsto T^w_\beta\}$;
   $C'' := \mathsf{Cl}(C')$;
   $\widehat{C} := \mathsf{At}(C'')$;
   if $\widehat{C} \models_L \alpha_w \leq^w \beta_w$ then ACCEPT else REJECT;

Figure 11.2: Improved PTIME algorithm for leaf–entailment.

dress in both $\Theta_C(\alpha)$ and $\Theta_C(\beta)$. Then the leaf entailment problem $C \models^w \alpha \leq \beta$ in structurally ordered finite trees over C–shaped valuations is decidable in PTIME.

11.2 coNP–hardness of finite entailment

We prove that structural, finite entailment is coNP–hard, *i.e.*, deciding the complementary problem $C \not\models \alpha \leq \beta$ is NP–hard. Moreover, we show that this already holds in the presence of a single binary covariant type constructor (that is, including $\times$ but excluding $\to$.)

The intuitive reason why finite (non–)entailment is intractable can already be explained from Proposition 11.1.1. According to that proposition, deciding the *non*–entailment $C \not\models \alpha \leq \beta$ comes down to checking the condition

$$\exists w \in \mathsf{Lf}(\Theta_C(\alpha)).\ C \not\models^w \alpha \leq \beta \tag{11.2}$$

assuming here that α and β are matching in C (otherwise the entailment trivially does not hold.) Since there may be exponentially many distinct leaf addresses in $\Theta_C(\alpha)$ in the worst case (recall Example 2.3.7), it is a relatively short step to the idea that such addresses might be used to encode propositional truth valuations, thereby reducing the propositional satisfiability problem SAT (see [30]) to the problem of *non*–entailment. The reduction is given in the proof of the following theorem. Example 11.2.2 below gives a pictorial view of the reduction, which should make it very easy to understand.

Theorem 11.2.1 *For any non-trivial lattice L, the predicate $C \models \alpha \leq \beta$ over finite, structurally ordered trees is hard for coNP under log-space reductions. Moreover, this holds in the presence of a single binary, co-variant type constructor.*

PROOF In the following construction, we assume only the type constructor $\times$. Moreover, since L is assumed to be a non-trivial lattice, we can assume that there are two elements $\bot, \top \in L$ such that $\bot \neq \top$ and $\bot \leq_L \top$.

Fix two distinct variables α and β. Let NENT be the problem:

- Given C, decide whether $C \not\models \alpha \leq \beta$

We reduce SAT (propositional satisfiability, [30]) to NENT. This shows that NENT is NP-hard, which in turn shows that the problem of deciding $C \models \alpha \leq \beta$ is coNP-hard.

The basic idea is that an address $w \in \{f, s\}^*$ defines a truth assignment of an instance of SAT. Given n variables $x_1, \ldots, x_n$ containing all the propositional variables occurring in an instance of SAT, we say that an address $w = a_1 \ldots a_n \in \{f, s\}^n$ of length n defines a truth assignment T_w as follows: $T_w(x_i) =$ true if $a_i = f$ and $T_w(x_i) =$ false if $a_i = s$, so f means *true* and s means *false*. Henceforth we shall think of addresses both as such and as the truth assignments they induce in this fashion.

Let us assume now that we are given an instance I of SAT, which is a set of clauses $\{C_1, \ldots, C_m\}$ over the propositional variables $x_1, \ldots, x_n$. Each clause is a finite disjunction of *atoms* A. An atom is a propositional variable or its negation. Each clause defines a set of truth valuations which *falsify* the clause; we call this set the *exclusion set* of the clause. Then, a truth valuation $T \in \{f, s\}^n$ satisfies I, if and only if T is not in the exclusion set of any clause in I.

Given I, we construct an instance $C(I)$ of NENT (that is, a constraint set) with the following intuition. For every clause $C_i = A_1 \vee \ldots \vee A_k$ in I we build a set of constraints that *excludes* exactly every address w that (when read as a truth assignment) *falsifies* the clause. By "excluding" we mean that every such address w satisfies

$$C(I) \models^w \alpha \leq \beta$$

so that, by Proposition 11.1.1, w is *not* a witness that the NENT-problem has a positive answer. This is done by making sure that w becomes a leaf address for $\Theta_{C(I)}(\alpha)$ and $\Theta_{C(I)}(\beta)$ in $C(I)$ such that $\Theta_{C(I)}(C(I))$ implies

the inequalities $\alpha_w \leq \bot$ and $\top \leq \beta_w$, with $\alpha_w = \Theta_{C(I)}(\alpha)(w)$ and $\beta_w = \Theta_{C(I)}(\beta)(w)$. Furthermore, our construction is such that $\alpha_w \leq \beta_w$ is never deducible from $C(I)$. This means that existence or nonexistence in $C(I)$ of an address w such that $\alpha_w \leq \bot$ and $\top \leq \beta_w$ determines whether $C(I) \not\models \alpha \leq \beta$ or $C(I) \models \alpha \leq \beta$.

Let us now describe in detail the construction of the set $C(I)$. We assume that we have m clauses $C_1, \ldots, C_m$ with a total of n propositional variables $x_1, \ldots, x_n$. The exclusion set of each clause C_i can be described by a unique *address pattern* $P_i \in \{f, s, \#\}^n$. An address pattern defines the set of addresses that arise by replacing $\#$ arbitrarily by either f or s. For example, $s\#\#fs\#\#\#$ is the address pattern that represents the falsifying truth valuations for the clause $x_1 \vee \neg x_4 \vee x_5$ for $n = 8$.

Given an address pattern P we define the constraint set $\mathcal{C}^+(P, \gamma)$ as follows:

$$\begin{aligned}
\mathcal{C}^+(fP', \gamma) &= \{\delta \times \delta' \leq \gamma\} \cup \mathcal{C}^+(P', \delta) \quad (\delta, \delta' \text{ new}) \\
\mathcal{C}^+(sP', \gamma) &= \{\delta \times \delta' \leq \gamma\} \cup \mathcal{C}^+(P', \delta') \quad (\delta, \delta' \text{ new}) \\
\mathcal{C}^+(\#P', \gamma) &= \{\delta \times \delta \leq \gamma\} \cup \mathcal{C}^+(P', \delta) \quad (\delta \text{ new}) \\
\mathcal{C}^+(\epsilon, \gamma) &= \{\top \leq \gamma\}
\end{aligned}$$

Similarly, we define $\mathcal{C}^-(P, \gamma)$: This is done by *reversing* the inequalities in the first three clauses for $\mathcal{C}^+$ above and replacing the last clause by $\mathcal{C}^-(\epsilon, \gamma) = \{\gamma \leq \bot\}$.

Let P_i be the address pattern that falsifies clause C_i. The constraints generated for C_i are

$$\mathcal{C}_i = \mathcal{C}^+(P_i, \beta) \cup \mathcal{C}^-(P_i, \alpha)$$

The constraints $C(I)$ generated for I is the union of the constraints generated for the individual clauses in I,

$$C(I) = \bigcup_{i=1}^{m} \mathcal{C}_i$$

Example 11.2.2 and Figure 11.3 illustrate this construction.

Now, to see that $C(I)$ is a reduction of SAT to NENT we have to check:

$$I \text{ is satisfiable if and only if } C(I) \not\models \alpha \leq \beta \tag{11.3}$$

Suppose first that I is satisfiable. Then there is a truth valuation $w \in \{f,s\}^n$ satisfying I. There is therefore no pattern P_i defining the exclusion set of C_i such that w matches P_i. It then follows by construction of $C(I)$ that the variables α_w and β_w (defined as above) will be unconstrained in the expansion of $C(I)$ under a most general matching substitution. Using Lemma 2.3.10 together with Theorem 8.1.2 on $C(I)^\flat$, it is easy to verify that $C(I) \not\models^w \alpha \leq \beta$ must hold, and hence $C(I) \not\models \alpha \leq \beta$. On the other hand, if $C(I) \not\models \alpha \leq \beta$, then there must be a leaf address w for $\Theta_{C(I)}(\alpha)$ and $\Theta_{C(I)}(\beta)$ such that $C(I) \not\models^w \alpha \leq \beta$ is true. By construction of $C(I)$, it is easy to see that this implies that there can be no pattern P_i such that w matches P_i, hence w is a valuation which is not in the exclusion set of any of the clauses in I, and therefore w must satisfy I. □

We end the section by illustrating the reduction pictorially, in the following example.

Example 11.2.2 Suppose we have an instance of SAT with $n = 5$ and consider the clause $\neg x_2 \vee x_4$. The address pattern defining the exclusion set of this clause is $P = \#f\#s\#$. The constraint set $\mathcal{C}^+(P,\beta)$ is computed as follows:

$$\begin{array}{lcl}
\mathcal{C}^+(P,\beta) & = & \{\delta_1 \times \delta_1 \leq \beta\} \cup \mathcal{C}^+(f\#s\#,\delta_1) \\
\mathcal{C}^+(f\#s\#,\delta_1) & = & \{\delta_2 \times \delta_3 \leq \delta_1\} \cup \mathcal{C}^+(\#s\#,\delta_2) \\
\mathcal{C}^+(\#s\#,\delta_2) & = & \{\delta_4 \times \delta_4 \leq \delta_2\} \cup \mathcal{C}^+(s\#,\delta_4) \\
\mathcal{C}^+(s\#,\delta_4) & = & \{\delta_5 \times \delta_6 \leq \delta_4\} \cup \mathcal{C}^+(\#,\delta_6) \\
\mathcal{C}^+(\#,\delta_4) & = & \{\delta_7 \times \delta_7 \leq \delta_6\} \cup \mathcal{C}^+(\epsilon,\delta_7) \\
\mathcal{C}^+(\epsilon,\delta_7) & = & \{\top \leq \delta_7\}
\end{array}$$

The constraints $\mathcal{C}^-(P,\alpha)$ are obtained by reversing these inequalities (using, of course new variables) and exchanging the last inequality with $\delta_7' \leq \bot$.

In order to get a pictorial view of this process, it is instructive to note that inequalities are only needed at two points, namely in the first step (introducing β, resp. α, into the system) and in the last step (introducing $\top$, resp. $\bot$, into the system); the remaining inequalities might as well have been taken as equalities. As a consequence, this construction can be regarded as constructing two trees, t_α and t_β, and the inequalities

$$\begin{array}{c}
\alpha \leq t_\alpha \\
t_\beta \leq \beta \\
t_\alpha(w) \leq \bot \\
\top \leq t_\beta(w)
\end{array}$$

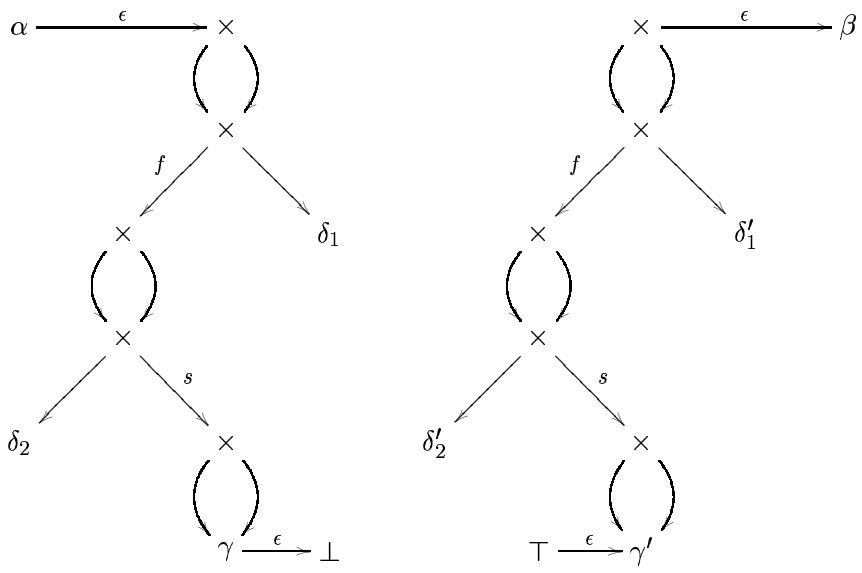


Figure 11.3: Constraints generated for the pattern $\#f\#s\#$ for $\neg x_2 \vee x_4$ with $n = 5$

where w ranges over address matching P.

The trees t_α and t_β may be exponentially large in n, but they can be defined by n inequalities, because each constraint generated for $\#$ repeats a variable. Pictorially, this can be made clear by using sharing in a graph representation of the constraint set. This is done in Figure 11.3 for the example above (the picture uses other variable names than the example above.) □

11.3 coNP upper bound for finite entailment

We show that the non–entailment problem, $C \not\models \alpha \leq \beta$, can be solved non–deterministically in polynomial time. Proposition 11.1.1 with the characterization (11.2) again yields the basic insight: in order to decide non–entailment, we may non–deterministically guess an appropriate address $w \in \mathbf{A}^*$ such that the condition (11.2) is witnessed by w.

The reason why this observation yields an NP–algorithm for non–entailment

is that the representation result in Theorem 10.2.3 shows that non–entailment over *C–shaped valuations* (recall Definition 10.2.1) has a succinct witness, in the form of an address $w \in \mathbf{A}^*$ of length at most n satisfying (11.1). Let us now explain this in more detail.

Outline of coNP–algorithm

To recall from Theorem 10.2.3, we know that, if α and β are *non–trivially related in* C (recall Definition 10.2.2), then entailment can be decided by deciding entailment restricted to C–shaped valuations. This restricted version of entailment, in turn, has succinct (in fact, linearly bounded) witnesses, because any value $v(\alpha)$ under a satisfying C–shaped valuation v must match $\Theta_C(\alpha)$; the depth of any tree $\Theta_C(\alpha)$, in turn, is bounded by n (the size of C), because $\Theta_C(\alpha)$ matches the value $U_C(\alpha)$ which can be computed and (graphically) represented in linear time. Hence, for non–trivially related α and β, restricted entailment can be decided by guessing and verifying a witness $w \in \mathbf{A}^*$ of size linearly bounded in the size of C; since w is linearly bounded, it can be guessed non–deterministically in polynomial time, and since leaf entailment is in PTIME (by Proposition 11.1.3), the verification step can be performed in PTIME. Finally, as is argued below, deciding whether α and β are non–trivially related can be done deterministically in polynomial time, and deciding entailment in the case where α and β are trivially related can be done deterministically in polynomial time.

We go on to describe this algorithm in detail. The description of the algorithm is divided into two parts, a pre–processing phase and a main step, shown in Figure 11.4 and Figure 11.5, respectively. Input to the algorithm is a constraint set C and variables α, β. The algorithm *accepts* if and only if the *non–entailment* $C \not\models \alpha \leq \beta$ holds over finite trees ordered structurally.

Pre–processing (Figure 11.4)

Figure 11.4 defines a pre–processing phase which performs a number of checks. Once this phase has been completed, the algorithm performs the operations described in Figure 11.5, unless the pre–processing phase already results in acceptance or rejection.

The first part checks that C is satisfiable. This check is implemented by checking weak unifiability and ground consistency, and correctness of this follows from Theorem 2.4.2 proven by Tiuryn ([71], Theorem 5), that satisfiability over finite structural trees generated from a lattice of base types

is equivalent to consistency. The first of these checks is performed by unification, and the second is performed by computing the closure $\mathsf{Cl}(C)$; both checks can therefore be performed deterministically in polynomial time.

The second part assumes (by the previous part) that C is satisfiable and checks that α and β occur non–trivially in C. If not, the algorithm accepts (*i.e.*, non–entailment is accepted.) Correctness follows from Lemma 10.2.4. The second step of this check determines whether or not the shapes of α and β in C are the same. This check can be performed by computing $U^\star_C(\alpha)$ and $U^\star_C(\beta)$, where $U^\star$ is the unifier U_C with all variables collapsed to the same element (arbitrarily chosen to be $\star$).

The third step determines whether it is the case that one but not both of $\Theta_C(\alpha)$ or $\Theta_C(\beta)$ has a leaf variable that is equivalent to a constant in C and hence forced to be mapped to a constant; this check can be performed in linear time by inspecting the unification graph of U_C (of size linear in the size of C) to see whether $\star$ is present in one (but not both) of the equivalence classes of a leaf of $U_C(\alpha)$ and $U_C(\beta)$.

Main step (Figure 11.5)

The second phase of the algorithm can assume that C is satisfiable and that α and β occur non–trivially in C, by the pre–processing phase. The second phase, shown in Figure 11.5, non–deterministically checks non–entailment over C-shaped valuations. Proposition 11.1.1 and Proposition 11.1.3 show that this phase is correct.

In this phase, we non–deterministically guess a leaf–address w in $U_C(\alpha)$, of length at most linear in the size of C. The leaf–addresses in $U_C(\alpha)$ are just the leaf–addresses in both $\Theta_C(\alpha)$ and $\Theta_C(\beta)$: α and β are matching in C, since α and β occur non–trivially in C.

Once a candidate witness w has been guessed, we verify that w is indeed a witness of non–entailment, by verifying that the corresponding leaf–entailment fails; correctness of this test follows from Proposition 11.1.1, and the test is in PTIME by Proposition 11.1.3.

We can now conclude from the coNP–algorithm together with Theorem 11.2.1 that we have

Theorem 11.3.1 *The problem of subtype entailment is coNP–complete for structurally ordered finite trees over a lattice of base types. Moreover, this holds in the presence of a single binary covariant type constructor.*

By adding suitable parts of the pre–processing step shown in Figure 11.4 to the leaf entailment algorithm shown in Figure 11.2, it is easy to see that we can drop the restrictions of Proposition 11.1.3 and get

Corollary 11.3.2 *The leaf entailment problem $C \models^w \alpha \leq \beta$ in structurally ordered finite trees is decidable in PTIME.*

As was said in Section 11.1, the PTIME leaf entailment algorithm can be improved by using a more subtle generalization of Theorem 8.1.2, which will eventually characterize structural recursive entailment. Once that has been done, we shall also obtain an improved coNP–algorithm for entailment in structural finite trees. As said, we postpone this until Chapter 12 in order to avoid too many tedious repetitions.

Given C, α, β, check that C is satisfiable (consistent). If C is inconsistent, reject.

1. (Check for weak unifiability) Compute U_C (most general unifier admitting infinite terms). If unification fails, REJECT.
2. (Check for ground consistency) Compute $\mathsf{Cl}(C)$. If $\mathsf{Cl}(C)$ is not ground consistent, REJECT.

Given consistent C, α and β. Check that α and β are non–trivially related in C. Non–entailment $C \not\models \alpha \leq \beta$ is accepted, if α and β are trivially related.

1. (Occurrence check) Check that both α and β occur in C; if not, then ACCEPT.
2. (Match check) Check that $U_C^\star(\alpha) = U_C^\star(\beta)$; if not, then ACCEPT;
3. (Constant constraint check) If $\star = U_C(\alpha)(w)$ and $\star \neq U_C(\beta)(w)$, or $\star \neq U_C(\alpha)(w)$ and $\star = U_C(\beta)(w)$, for some leaf address w in $U_C(\alpha)$ and $U_C(\beta)$, then ACCEPT

Figure 11.4: coNP algorithm for entailment. Preprocessing phase.

Given C, α and β with C consistent and α, β occurring non–trivially in C. The algorithm accepts if and only if $C \not\models \alpha \leq \beta$ over C–shaped valuations in finite trees ordered structurally.

1. Guess a leaf–address w in $U_C(\alpha)$;
2. If $C \not\models^w \alpha \leq \beta$ then ACCEPT;

Figure 11.5: coNP algorithm for entailment. Main step.

Chapter 12

Structural recursive entailment

In this chapter, we prove that the entailment problem for structural recursive subtyping is PSPACE–complete. The structure $\mathcal{T}_\Sigma$ is now fixed to be $\mathcal{T}_\Sigma[s]$, structurally ordered finite and infinite trees. The lower bound proof is an extension of the PSPACE lower bound for the non–structural case (Chapter 9). The upper bound is based on a generalization of our characterization of atomic entailment (Chapter 8).

12.1 PSPACE–hardness for recursive entailment

In the remainder of this chapter we will prove that the entailment problem $C \models \alpha \leq \beta$ over $\mathcal{T}_\Sigma[s]$ is PSPACE–hard. First, let us observe that the reduction used to prove PSPACE–hardness for non-structural recursive subtyping in Section 9.3 does not transfer directly to the structural case. To see the difference, let A be the NFA shown in Figure 12.1, with start state q_0 and all states accepting, and consider the constraint set C_A as defined in Section 9.3. A accepts (in addition to the empty string) just the strings over $\{f, s\}$ of either one of the forms ff^n, fs^n, ss^n, sf^n, $n \geq 0$. Thus, we certainly have $L(A) \neq \{f, s\}^*$. However, by weak unification of C_A (*i.e.*, solving unification for E_{C_A}) the reader can easily verify that any solution to C_A must map α_0 to the complete, infinite binary tree $t^\infty = \mu\gamma.\gamma \times \gamma$. What happens under unification in C_A corresponds to collapsing the states q_1, q_2, q_3, q_4 of A into a single state. This means that, for this particular set C_A, we have $C_A \models \alpha_0 \leq t^\infty$ but not $L(A) = \{f, s\}^*$, and our previous

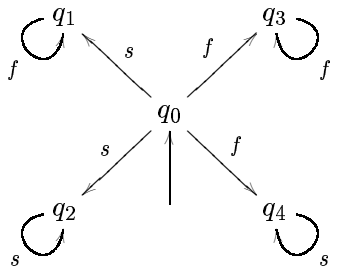


Figure 12.1: NFA accepting $f(f^* \mid s^*) \mid s(s^* \mid f^*)$.

reduction is seen to be incorrect for the structural case.

Even though we cannot use exactly the same technique as was used in the non–structural case, it turns out that non-determinism can still be faithfully represented by structural constraints, using the basic idea of encoding automata by constraint-automata. Only, one has to elaborate the construction, and we now outline how this is done.

The basic idea is to allow the variable α_0 (representing the start state of a given automaton) to have one particular, fixed shape, which will be compatible with all inequalities generated from the automaton A. As can be seen from our example, that shape must presumably be infinite. Define the infinite trees $t^\bot$ and $t^\top$ by

$$t^\bot = \mu\gamma.(\gamma \times \bot) \otimes (\gamma \times \bot) \text{ and } t^\top = \mu\gamma.(\gamma \times \top) \otimes (\gamma \times \top)$$

(Here we have used $\otimes$ as another name for $\times$, in order to single out special occurrences of $\times$, because the constructor $\times$ will play two distinct rôles in the reduction in a way that will be explained later.) Our reduction, then, will be such that in all cases α_0 will satisfy

$$\alpha_0 \leq t^\top \tag{12.1}$$

The inequality in (12.1) implies by Lemma 2.1.1 that we have in fact

$$t^\bot \leq \alpha_0 \leq t^\top \tag{12.2}$$

and the encoding of an automaton A will only depend upon whether or not the leaves of $v(\alpha_0)$ must be $\bot$, for any solution v to C_A. More specifically, the relation (12.2) implies (via Lemma 2.1.1) that, for any solution v to C_A,

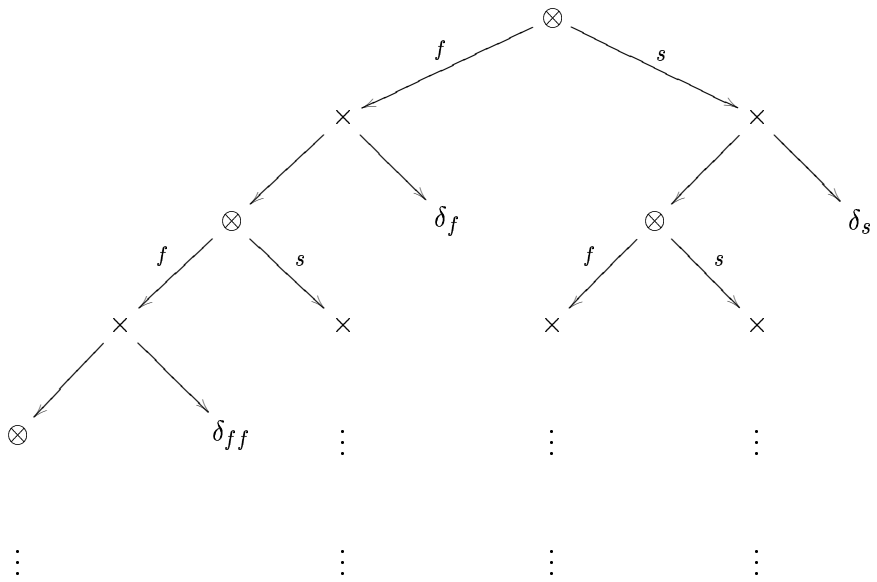


Figure 12.2: Tree s_0 showing the shape of α_0

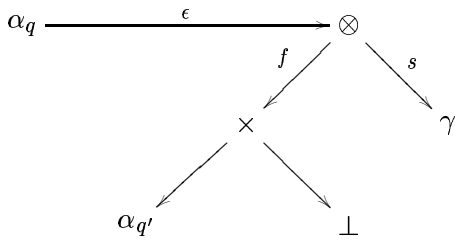


Figure 12.3: Constraint automaton corresponding to NFA transition $q \mapsto^f q'$

$v(\alpha_0)$ must have the shape of the tree s_0 shown in Figure 12.2, where the leaves constitute an infinite collection of distinct variables δ_w which must satisfy $\bot \leq \delta_w \leq \top$, and so, in particular, each leaf variable δ_w must be mapped to a constant in any solution for α_0. The annotations on the tree used in Figure 12.2 will be explained later.

We now give the definition of a constraint set C_A constructible in logspace from a given NFA A. As before, we construct C_A from sets C_k corresponding to each k'th simple transition in A. The C_k are defined as follows:

- If the k'th simple transition in A is $q_i \mapsto^f q_j$, then
$$C_k = \{\alpha_i \leq (\alpha_j \times \bot) \otimes \gamma_k\}$$
- If the k'th simple transition in A is $q_i \mapsto^s q_j$, then
$$C_k = \{\alpha_i \leq \gamma_k \otimes (\alpha_j \times \bot)\}$$
- If the k'th simple transition in A is $q_i \mapsto^\epsilon q_j$, then
$$C_k = \{\alpha_i \leq \alpha_j\}$$

Here the γ_k are fresh variables. The set C_A is defined to be the inequality shown in (12.1) together with the union of all the C_k (the element $t^\top$ can be defined by regular equations.) Figure 12.3 shows the constraint automaton generated from a transition $q \mapsto^f q'$. We have not given the inequalities in simple form, but we can think of C_A as simple, by trivial transformation.

Lemma 12.1.1 *Let A be a prefix–closed NFA over $\{f, s\}$. Then*
$$L(A) = \{f, s\}^* \text{ if and only if } C_A \models \alpha_0 \leq t^\bot$$

PROOF We first give an intuitive explanation of the construction of C_A. Looking at Figure 12.2, the reader will notice that going down a branch of s_0, the products alternate between $\otimes$ and $\times$ (which, to recall, are really all the same product constructor, only named differently to distinguish occurrences.) Let us call all occurrences labelled $\otimes$ *primary products* and all those labelled $\times$ *secondary*. Then one has that primary products are located at the addresses in the language [1]

$$L_\otimes = [(f \mid s)f]^*$$

and that leaf variables (denoted δ_w in Figure 12.2) are located at the addresses in the language

$$L_\Delta = L_\otimes \cdot [(f \mid s)s]$$

Secondary products only serve the purpose of attaching a pair of leaf variables δ_w to each primary product (sitting at addresses fs and ss from the primary product.)

In order to explain the significance of this, we need a few definitions. If $w = a_1 a_2 \ldots a_n$ is a word in $\{f, s\}^*$, let

$$w\|f = a_1 f a_2 f \ldots a_n f$$

i.e., $w\|f$ is w interleaved with f (and we take $\epsilon\|f = \epsilon$). Then define, for $w = w'a$, $a \in \{f, s\}$, the operation Δ by setting

$$\Delta(w) = (w'\|f)as$$

Then, for any $w \in \{f, s\}^* \setminus \{\epsilon\}$, one has $\Delta(w) \in L_\Delta$. The idea now is that the inequality (12.2) forces $\Theta_{C_A}(\alpha_0)$ to have the fixed shape shown in Figure 12.2, where δ_w is a fresh leaf variable sitting at address $\Delta(w)$. The set C_A, in turn, is constructed in such a way (see Figure 12.3) that a non–empty word $w \in L(A)$ will give rise to the constraint $\delta_w \leq \bot$ (implied in the expansion of C_A under Θ_{C_A}). This way, the variables δ_w sitting under the secondary products in $\Theta_{C_A}(\alpha_0)$, serve as markers that a transition on f or s out of the primary product has been taken. Since there is such a marker variable δ_w at address $\Delta(w)$ for each non–empty word w, constraints on the fixed shape of α_0 can represent the fact that an arbitrary non–empty word w is accepted by A.

[1] In our notation we identify a regular expression with the language denoted by it. The operator $\cdot$ is concatenation of languages.

The intention of our encoding can now be stated precisely. We say that an address w is *marked* if and only if the following two properties hold for all valuations v satisfying C_A:

1. $\Delta(w) \in \mathcal{D}(v(\alpha_0))$, and
2. $v(\alpha_0)(\Delta(w)) \leq \bot$

Then we claim:

$$w \in L(A) \text{ with } w \neq \epsilon \text{ if and only if } w \text{ is marked} \tag{12.3}$$

First notice that, because (12.2) always holds by construction of C_A, the first condition above holds for every address $w \neq \epsilon$. Suppose then that $w \neq \epsilon$, $w \in L(A)$. Then, by construction of C_A, there is a w-transition in $\mathcal{A}_N^{C_A}$. By expanding α_0 to the shape shown in Figure 12.2, it is easy to see that this transition in $\mathcal{A}_N^{C_A}$ implies the condition $\delta_w \leq \bot$ on the variable δ_w in the expansion $\Theta_{C_A}(\alpha_0)$ (δ_w sitting at address $\Delta(w)$ in the expansion of α_0). It follows that any valuation v satisfying C_A must also satisfy $v(\alpha_0)(\Delta(w)) \leq \bot$, showing that w is marked. Conversely, assume $w \notin L(A)$. Then, *because A is prefix closed*, there can be no w-transition in A starting from q_0 at all. By construction of C_A, there is therefore no $\Delta(w)$-transition in $\mathcal{A}_N^{C_A}$. It is easy to see that this implies that the variable δ_w sitting at address $\Delta(w)$ in the expansion of α_0 (according to Figure 12.2) will not be bounded by $\bot$ in the expanded set $\Theta_{C_A}(C_A)$. In particular, the satisfying valuation $v_\wedge$ of Theorem 10.4.2 must have $v_\wedge(\alpha_0)(\Delta(w)) = \top$, showing that w is not marked. We have now shown that (12.3) is true.

We can now prove the lemma from (12.3). First observe that the value of $v(\alpha_0)(w)$ is fixed to be the product constructor for any address $w \notin L_\Delta$, whenever v satisfies C_A. Therefore, one has $v(\alpha_0) \leq t^\bot$ if and only if $v(\alpha_0)(\Delta(w)) \leq \bot$ for all $w \in \{f,s\}^* \setminus \{\epsilon\}$. If $L(A) = \{f,s\}^*$ and $v \models C_A$, then all addresses in $w \in \{f,s\}^* \setminus \{\epsilon\}$ are marked, by (12.3), and hence $v(\alpha_0) \leq t^\bot$, showing $C_A \models \alpha_0 \leq t^\bot$. Conversely, if $L(A) \neq \{f,s\}^*$, then we can assume that $w \notin L(A)$ for some $w \neq \epsilon$, because A is prefix closed and any non–empty prefix closed automaton must accept ϵ. Then, by (12.3), w is not marked, and hence $v(\alpha_0)(\Delta(w)) \not\leq \bot$ for some v satisfying C_A, showing $C_A \not\models \alpha_0 \leq t^\bot$. □

The above lemma shows that the construction C_A is a logspace reduction of CLOSED-UNIV to structural entailment over infinite trees, and we therefore have:

Theorem 12.1.2 *The problem of structural, recursive subtype entailment is PSPACE–hard.*

12.2 Witnessing non–entailment

Beginning this section and continuing through Section 12.4, we will show that the problem of structural recursive subtype entailment can be solved in PSPACE. We give a non–deterministic algorithm for deciding *non–entailment* (*i.e.*, deciding $C \not\models \alpha \leq \beta$?) that runs in polynomial space. The PSPACE result follows, because PSPACE = NPSPACE = coNPSPACE by Savitch's Theorem.

A starting point for the non–deterministic algorithm is the following obvious observation. The entailment $C \models \alpha \leq \beta$ does *not* hold if and only if there exist trees t_1 and t_2 in $\mathcal{T}_\Sigma$ such that

(i) $t_1 \not\leq t_2$, and

(ii) $C[t_1/\alpha, t_2/\beta]$ is satisfiable

However, while t_1 and t_2 could be infinite in the general, recursive case, there will always be a *finite witness* in $\mathbf{A}^*$ of the fact (i) that $t_1 \not\leq t_2$; this witness is a finite address $w \in \mathcal{D}(t_1) \cap \mathcal{D}(t_2)$ such that $t_1(w) \not\leq^w t_2(w)$. This means that, in case the entailment $C \models \alpha \leq \beta$ does not hold, then we can always choose finite and "thin" trees to witness this fact. More precisely, if t_1 and t_2 are any trees in $\mathcal{T}_\Sigma$ satisfying (i) and (ii) and w is a witness of (i), then we can define finite trees t_i^w to be smallest trees (having least tree domains) such that $w \in \mathcal{D}(t_i^w)$ and $t_i^w(w) = t_i(w)$. It is easy to see that this can always be done in such a way that (i) and (ii) are equivalent to

(i') $t_1^w \not\leq t_2^w$, and

(ii') $C[t_1^w/\alpha, t_2^w/\beta]$ is satisfiable

As an additional simplification of the problem, we can read "consistent" instead of "satisfiable" in the conditions (ii) and (ii'), since consistency and satisfiability are equivalent (Theorem 9.1.5). An idea then, for a non–deterministic algorithms to decide non–entailment, would be to guess a witness w and somehow verify that conditions such as (i') and (ii') are satisfied for w.

The serious complication with this idea is that a witness might be exponentially long, so we cannot actually construct the witness explicitly, remaining within PSPACE. This is quite analogous to the situation for the non–universality problem for NFA's (does a given automaton A *not* accept all strings over its alphabet?). A witness to the non–universality property is a word w not accepted by the automaton A, and such a word might be exponentially long in the size of A. The solution, in the NPSPACE algorithm for non–universality [1], is to guess the witness w bit by bit, forgetting all but the last bit of the witness; the reason this works is that the algorithm can simulate the behaviour of the automaton sufficiently, using this bit together with a subset of the state set of the automaton; the state set contains the states reachable from the start state on the witness guessed so far; the last bit of the witness is sufficient to compute the next state set. Our NPSPACE algorithm for deciding non–entailment will, at the bottom of it, be very similar to this. Our algorithm will construct its witness $w \in \mathbf{A}^*$ by keeping track of the last bit guessed together with subsets of the nodes in the constraint graph. Intuitively, the node sets are used to simulate relevant information about the closure $\mathsf{Cl}(C[t_1^w/\alpha, t_2^w/\beta])$.

12.3 Characterization

Our algorithm is founded on a characterization theorem, which we proceed to present. In order to state the characterization, we fix a constraint set C and consider the constraint graph $\mathcal{G}_C = (V, E)$ and define a relation

$$R \subseteq V \times V \times \{\Downarrow, \Uparrow\} \times \mathbf{A}^*$$

Intuitively, this relation captures properties of the closure of the constraint set C. In order to define R, let $w \in \mathbf{A}^*$, and let the relation $\mapsto_\epsilon^w$ denote $\mapsto_\epsilon$ (the ϵ–transitions in $\mathcal{G}_C$) or the reverse relation, according to the polarity of w ($\pi(w) = 0$ and $\pi(w) = 1$, respectively), and we let $\mapsto_\epsilon^{\overline{w}}$ denote the reverse of $\mapsto_\epsilon^w$. This notation will be used to make sure that contravariance is respected by the algorithm. Intuitively, the relation R holds of $(v, v', \Uparrow, w)$ if there is a w-path in $\mathcal{G}_C$ from v to v' where ϵ-edges are followed in reversed direction according to the polarity of w. The relation R is defined by induction on $w \in \mathbf{A}^*$, as follows:

- $w = \epsilon$:

$$R(v, v', \Uparrow, w) \quad \Leftrightarrow \quad v \mapsto_\epsilon v'$$

$$R(v, v', \Downarrow, w) \quad \Leftrightarrow \quad v' \mapsto_\epsilon v$$

- $w = w'a$:

$$\begin{array}{ll} R(v, v', \Uparrow, w) & \Leftrightarrow \\ \exists v_1, v_2.\ R(v, v_1, \Uparrow, w') & \wedge \\ v_1 \mapsto_a v_2 & \wedge \\ v_2 \mapsto_\epsilon^w v' & \end{array}$$

$$\begin{array}{ll} R(v, v', \Downarrow, w) & \Leftrightarrow \\ \exists v_1, v_2.\ R(v, v_1, \Downarrow, w') & \wedge \\ v_1 \mapsto_a v_2 & \wedge \\ v_2 \mapsto_\epsilon^{\overline{w}} v' & \end{array}$$

Let

$$\Uparrow_C^w (v) = \{v' \in V \mid R(v, v', \Uparrow, w)\}$$

and

$$\Downarrow_C^w (v) = \{v' \in V \mid R(v, v', \Downarrow, w)\}$$

and define

$$\alpha \preceq_C^w \beta$$

to hold if and only if there exists a prefix w' of w such that

$$\Uparrow_C^{w'} (\alpha) \cap \Downarrow_C^{w'} (\beta) \neq \emptyset$$

Recall from Definition 11.1.2 that, for a word $w \in \mathbf{A}^*$, a *w-template* is a finite term T^w such that $\mathcal{D}(T^w)$ is the least tree domain containing w and the leaves contain fresh, pairwise distinct variables. Now, we can assume that α and β have the same shape in C, *i.e.*, $s_C(\alpha) = s_C(\beta)$ (otherwise the entailment $C \models \alpha \leq \beta$ trivially does not hold if C is satisfiable). For a word w of maximal length (*i.e.*, a leaf address) in $s_C(\alpha)(= s_C(\beta))$ we let T_α^w and T_β^w be two distinct w-templates, having no variables in common with each other or C. Then we have the following lemma:

Lemma 12.3.1 *Let C be closed and let w be a leaf address in the common shape of α and β in C. Let $\alpha_w = T_\alpha^w(w)$, $\beta_w = T_\beta^w(w)$ and $C^+ = Cl(C[T_\alpha^w/\alpha, T_\beta^w/\beta)$. Then*

1. *$\alpha_w \leq^w \beta_w \in C^+$ if and only if $\alpha \preceq_C^w \beta$*
2. *$\alpha_w \leq^w b \in C^+$ if and only if $b \in \Uparrow_C^w (\alpha)$*

3. $b \leq^w \beta_w \in C^+$ *if and only if* $b \in \Downarrow^w_C (\beta)$

PROOF The implications from right to left are quite obvious, and can easily be proved by induction on the length of w. For the implications from right to left, we show how to prove the implication of the first claim of the lemma; the two other implications are proven by similar methods and the details are left out. To prove the implication from left to right of the first claim of the lemma, we first prove the following claim:

(∗) Let w' be a prefix of w and let $v^{w'}_\alpha$ and $v^{w'}_\beta$ be the vertices in $\mathcal{G}_{C^+}$ representing $T^w_\alpha(w')$ and $T^w_\beta(w')$ respectively, and let v be a vertex in $\mathcal{G}_C$. If $v^{w'}_\alpha \mapsto^{w'}_\epsilon v$ holds in $\mathcal{G}_{C^+}$, then $v \in \Uparrow^{w'}_C (\alpha)$, and if $v \mapsto^{w'}_\epsilon v^{w'}_\beta$ holds in $\mathcal{G}_{C^+}$, then $v \in \Downarrow^{w'}_C (\beta)$.

We show only the first part of the claim (concerning the situation where $v^{w'}_\alpha \mapsto^{w'}_\epsilon v$), since the other part is analogous. The claim is proven by induction in the length of w', and for the base case ($w' = \epsilon$), the transition $\alpha \mapsto_\epsilon v$ must already be present in $\mathcal{G}_{\mathsf{Cl}(C)}$, and hence $v \in \Uparrow^\epsilon_C (\alpha)$ is clear. For the inductive case, suppose that $w' = w''d$ (this case showing contravariance is representative and other cases are left to the reader). Clearly, the only way the transition $v^{w'}_\alpha \mapsto^{w'}_\epsilon v$ could get into $\mathcal{G}_{C^+}$ is by an application of the decomposition closure rule (possibly followed by a number of applications of the transitive closure rule). A schematic view of the situation is shown in Figure 12.5, where we assume that w' is negative (*i.e.*, $\pi(w') = 1$) – the situation where w' is positive is similar and left out. Let v_1 be the vertex shown as $\rightarrow^i$ in the figure and let v_2 be the vertex $\rightarrow^j$. Then, by induction hypothesis, we have $v_2 \in \Uparrow^{w''}_C (\alpha)$, *i.e.*, $R(\alpha, v_2, \Uparrow, w'')$ holds, with w'' positive. Then, by continuing from v_2 on a d-transition to v' and further along reversed ϵ-transitions to v, we arrive at a continuation of the w''-transition witnessing $R(\alpha, v_2, \Uparrow, w'')$ such that $R(\alpha, v, \Uparrow, w')$ holds. This ends the proof of the claim (∗).

Now, to prove the first claim of the lemma, assume that $\alpha_w \leq^w \beta_w \in C^+$, *i.e.*, there is a transition $\alpha_w \mapsto^w_\epsilon \beta_w$ in $\mathcal{G}_{C^+}$. Then either $\alpha \leq \beta$ is in $\mathsf{Cl}(C)$, or else there must be a prefix w' of w and a vertex v in $\mathcal{G}_C$ such that we have $v^{w'}_\alpha \mapsto^{w'}_\epsilon v$ and $v \mapsto^{w'}_\epsilon v^{w'}_\beta$ in $\mathcal{G}_{C^+}$, where $v^{w'}_\alpha$ (resp. $v^{w'}_\beta$) is the node in $\mathcal{G}_{C^+}$ representing $T^w_\alpha(w')$ (resp. $T^w_\beta(w')$). A typical situation falling under the latter case is shown in Figure 12.6 for illustration. In this case, it follows from the property (∗) that one has $v \in \Uparrow^{w'}_C (\alpha) \cap \Downarrow^{w'}_C (\beta)$, thereby proving $\alpha \preceq^w_C \beta$; in the former case where $\alpha \leq \beta \in \mathsf{Cl}(C)$, we have $\alpha \preceq^\epsilon_C \beta$. This

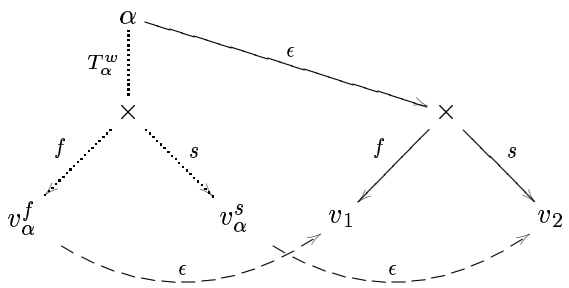


Figure 12.4: View of C and extension to C^+. Types added to C in C^+ are shown with dotted arcs. Transitions added under closure of C^+ are shown with dashed arcs. The w'-transitions in $\mathcal{G}_{C^+}$ from root of T_α^w to vertices in $\mathcal{G}_C$ commute with paths in $\mathcal{G}_C$ traced by $\Uparrow_C^w(\alpha)$

proves the first claim of the lemma. The two remaining claims are proven by the same technique, and details are left to the reader. □

Let $\bigwedge^w$ be the greatest lower bound operator in L, if $\pi(w) = 0$, and the least upper bound operator if $\pi(w) = 1$; dually, let $\bigvee^w$ denote least upper bound in the first case and greatest lower bound in the second case. If $S \subseteq V$ is a set of vertices in a constraint graph, then we define $\bigwedge^w(S) = \bigwedge^w(L \cap S)$, *i.e.*, the operation is taken on the set of vertices in S that represent constants from L (the operation $\bigvee^w(S)$ is defined analogously, of course).

Using the previous lemma, we can prove the following theorem which characterizes entailment over C-shaped valuations. The theorem is a generalization of Theorem 8.1.2 for the atomic case.

Theorem 12.3.2 *$C \models \alpha \leq \beta$ (over C-shaped valuations) if and only if one of the following conditions holds for every address w of maximal length in the common shape of α and β in C:*

(i) $\alpha \preceq_C^w \beta$ *or*

(ii) $\bigwedge^w \Uparrow_C^w(\alpha) \leq_L^w \bigvee^w \Downarrow_C^w(\beta)$

PROOF We begin by showing the implication $(i) \vee (ii) \Rightarrow C \models \alpha \leq \beta$ over C-shaped valuations. So assume that we have

(i) $\alpha \preceq_C^w \beta$ or

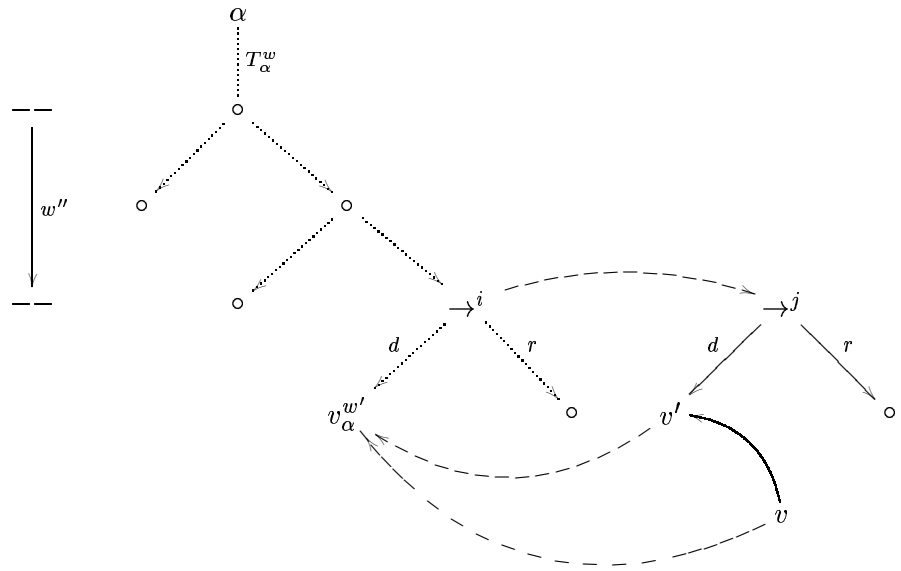


Figure 12.5: The transition $v^{w'}_\alpha \mapsto^{w'}_\epsilon v$ (with $w' = w''d$ and w' negative) added to $\mathcal{G}_{C^+}$ by an application of the decomposition closure rule followed by an application of the transitive closure rule. (The vertex $\to^i$ represents $T^w_\alpha(w')$.)

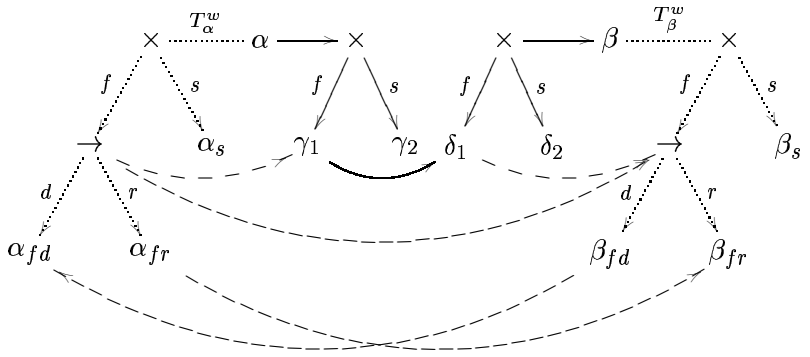


Figure 12.6: View of C and extension to C^+. Types added to C in C^+ are shown with dotted arcs. Transitions added under closure of C^+ are shown with dashed arcs. With $w = fd$ and $w' = f$ we have $\alpha_w \leq^w \beta_w$ in $\mathcal{G}_{C^+}$ and also $\Uparrow^{w'}_C(\alpha) \cap \Downarrow^{w'}_C(\beta) = \{\gamma_1, \delta_1\}$ in $\mathcal{G}_C$

(ii) $\bigwedge^w \Uparrow^w_C (\alpha) \leq^w_L \bigvee^w \Downarrow^w_C (\beta)$

for every $w \in \mathsf{Lf}(\Theta_C(\alpha)) = \mathsf{Lf}(\Theta_C(\beta))$. By Lemma 10.1.5, it is sufficient to show that $C^\flat \models [\alpha \leq \beta]_C$. Since we are only considering C-shaped valuations, this will follow from

$$C^\flat \models_L [\alpha \leq \beta]_C \tag{12.4}$$

because every C-shaped valuation on C factors through a corresponding valuation on $C^\flat$ in L (Lemma 10.1.5, first part.) In order to show (12.4), in turn, we must show

$$C^\flat \models_L \alpha_w \leq^w \beta_w \tag{12.5}$$

for all $w \in \mathsf{Lf}(\Theta_C(\alpha)) = \mathsf{Lf}(\Theta_C(\beta))$, and with $\alpha_w = \Theta_C(\alpha)(w)$, $\beta_w = \Theta_C(\beta)(w)$. So let w be an arbitrary address in $\mathsf{Lf}(\Theta_C(\alpha))$ (or $\mathsf{Lf}(\Theta_C(\beta))$). Suppose first that (by our assumption, (i)) $\alpha_w \preceq^w_C \beta_w$ holds. The trees $\Theta_C(\alpha)$ and $\Theta_C(\beta)$ induce w-templates, namely such that agree with $\Theta_C(\alpha)$ (resp. $\Theta_C(\beta)$) at the leaves of the latter. More precisely, let θ^w_α (resp. θ^w_β) be w-templates such that $\theta^w_\alpha(w') = \Theta_C(\alpha)(w')$ for all $w' \in \mathcal{D}(\theta^w_\alpha) \cap \mathsf{Lf}(\Theta_C(\alpha))$ (and similarly for θ^w_β). Then Lemma 12.3.1 implies that $\alpha_w \leq^w \beta_w \in \mathsf{Cl}(C[\theta^w_\alpha/\alpha, \theta^w_\beta/\beta])$, and therefore it is easily seen that we must have

$$C^\flat \vdash_L \alpha_w \leq^w \beta_w \tag{12.6}$$

On the other hand, suppose that the other possible case, (ii), holds, by our assumption. Recall that we define

$$\uparrow_C (\alpha) = \{b \in L \mid C \vdash_L \alpha \leq b\}$$

and

$$\downarrow_C (\alpha) = \{b \in L \mid C \vdash_L b \leq \alpha\}$$

Let $\uparrow^w_C = \uparrow_C$ if $\pi(w) = 0$ and otherwise (if $\pi(w) = 1$) set $\uparrow^w_C = \downarrow_C$ (and $\downarrow^w_C$ is defined likewise.) By (ii) we have $\bigwedge^w \Uparrow^w_C (\alpha) \leq^w_L \bigvee^w \Downarrow^w_C (\beta)$. Then, by Lemma 12.3.1, we have

$$\bigwedge^w \uparrow^w_{C^+} (\alpha_w) \leq^w_L \bigvee^w \downarrow^w_{C^+} (\beta_w)$$

with $C^+ = \mathsf{Cl}(C[\theta^w_\alpha/\alpha, \theta^w_\beta/\beta])$, where θ^w_α and θ^w_β are w-templates defined as before. Since $C^\flat$ clearly imposes at least the constraints on α_w and β_w as does C^+, it is easy to see that we must have

$$\bigwedge^w \uparrow^w_{C^\flat} (\alpha_w) \leq^w_L \bigvee^w \downarrow^w_{C^\flat} (\beta) \tag{12.7}$$

We conclude that either (12.6) or (12.7) must be the case. Then (12.5) follows from Theorem 8.1.2. We have now shown the desired implication.

We will now show the converse implication, $C \models \alpha \leq \beta \Rightarrow (i) \vee (ii)$. To prove this implication, assume that neither (i) nor (ii) holds for some address $w \in \mathsf{Lf}(\Theta_C(\alpha))(= \mathsf{Lf}(\Theta_C(\alpha)))$. Fix such an address w. We must show that $C \not\models \alpha \leq \beta$ over C-shaped valuations. Let T^w_α and T^w_β be w-templates and let $C^+ = \mathsf{Cl}(C[T^w_\alpha/\alpha, T^w_\beta/\beta])$. Let

$$\widehat{C}^+ = \{A \leq A' \mid A \leq A' \in C^+, \text{ and } A, A' \text{ are atoms}\}$$

It follows from Lemma 12.3.1 (together with the assumption that neither (i) nor (ii) is the case) that

(i') $\alpha_w \leq^w \beta_w \notin C^+$ and

(ii') $\bigwedge^w \uparrow^w_C (\alpha) \not\leq^w_L \bigvee^w \downarrow^w_C (\beta)$

It follows from Theorem 8.1.2 applied to the atomic constraint set $\widehat{C}^+$ that we have $\widehat{C}^+ \not\models_L \alpha_w \leq^w \beta_w$. Hence, for some $b_1, b_2 \in L$ with $b_1 \not\leq^w b_2$ we have $S(\widehat{C}^+)$ consistent with $S = \{\alpha_w \mapsto b_1, \beta_w \mapsto b_2\}$. But then we claim that

$$S(C^+) \text{ is consistent} \tag{12.8}$$

To see that (12.8) is true, consider that the closure of the set $S(C^+)$ can be obtained from C^+ (which is already closed) by applications of the transitivity rule alone (no decomposition rules need to be used, because α_w and β_w each have just a single occurrence in fresh templates, and α_w and β_w cannot be compared to any structured types in $S(C^+)$). It is therefore easy to see that consistency of $S(\widehat{C}^+)$ implies that (12.8) is true. By Theorem 10.4.2, it then follows that $S(C^+)$ is satisfiable. By Theorem 10.2.3, $S(C^+)$ is then satisfied by a C^+-shaped valuation, v. Then the valuation $\widehat{v} = v \circ \{\alpha \mapsto S(T^w_\alpha), \beta \mapsto S(T^w_\beta)\}$ is C-shaped and satisfies C. We have

$$\widehat{v}(\alpha)(w) = b_1 \not\leq^w b_2 = \widehat{v}(\beta)(w)$$

thereby showing $\widehat{v}(\alpha) \not\leq \widehat{v}(\beta)$. Hence, $\widehat{v}$ witnesses that $C \not\models \alpha \leq \beta$ over C-shaped valuations, as desired. □

12.4 PSPACE algorithm

Theorem 12.3.2 yields the central idea for our PSPACE–algorithm: Given a structural constraint set C, which is consistent (weakly unifiable and ground

consistent), and in which α and β occur non–trivially, we guess a path $w \in \mathcal{D}(\Theta_C(\alpha))$ and check that both conditions of Theorem 12.3.2 are violated. If so, we accept (which corresponds to reporting "not-entails"), otherwise we reject. More precisely, we iterate the following loop, with w initialized to ϵ:

1. If $\alpha \preceq_C^w \beta$ then reject.
2. Otherwise, if w is a leaf address in $\Theta_C(\alpha)$ and $\bigwedge^w \Uparrow_C^w (\alpha) \leq_L^w \bigvee^w \Downarrow_C^w (\beta)$, then reject.
3. Otherwise, if w is a leaf address then accept.
4. Otherwise, if w is not a leaf address, then guess $a \in \mathbf{A}$ such that $wa \in \mathcal{D}(\Theta_C(\alpha))$, set $w := wa$ and go to Step 1.

What essentially makes this a (nondeterministic) PSPACE algorithm for deciding non–entailment is that we do not actually need to store w, only $\Uparrow_C^w (\alpha)$ and $\Downarrow_C^w (\beta)$ because just these sets are required to perform the test for the conditions of Theorem 12.3.2. Furthermore, $\Uparrow_C^{wa} (\alpha)$ and $\Downarrow_C^{wa} (\beta)$ can be computed from a and $\Uparrow_C^w (\alpha)$ and $\Downarrow_C^w (\beta)$ in polynomial time and space (full details are given below). Thus the algorithm essentially requires only space for $\Uparrow_C^w (\alpha)$ and $\Downarrow_C^w (\beta)$, which is polynomial in the number of vertices and thus also in the size of the input; in addition, we need to know the (leaf) addresses in $\Theta_C(\alpha)(= \Theta_C(\beta))$, but this information can be obtained by computing the unifier $U_C(\alpha)$ (stored as a graph structure of size almost–linear in the size of the constraint set.)

We now prepare to give the algorithm in full detail. We begin by showing how to compute the sets $\Uparrow_C^{wa}$, $\Downarrow_C^{wa}$ from the sets $\Uparrow_C^w$, $\Downarrow_C^w$. Fix the constraint set C and its constraint graph $\mathcal{G}_C = (V, E)$. For $a \in \mathbf{A}$, let $F_a : V \to \wp(V)$ be given by

$$F_a = \{v' \in V \mid v \mapsto_a v'\}$$

and $F_\epsilon : \{0,1\} \times V \to \wp(V)$ by

$$F_\epsilon(b, v) = \{v' \in V \mid v \mapsto_\epsilon^b\}$$

Here, $b \in \{0,1\}$ is a bit which controls the direction of $\mapsto_\epsilon$ in the usual way. If $S \subseteq V$, we let

$$F_a(S) = \bigcup_{v \in S} F_a(v) \text{ and } F_\epsilon(b, S) = \bigcup_{v \in S} F_\epsilon(b, v)$$

Now, if $\pi(wa) = b$, then we can write (by definitions)

$$\Uparrow_C^{wa} = \{v' \in V \mid \exists v_1 \in \Uparrow_C^w (v).\ \exists v_2 \in V.\ v_1 \mapsto_a v_2 \wedge v_2 \mapsto_\epsilon^b v'\}$$

which shows that we have

$$\Uparrow_C^{wa} (v) = F_\epsilon(b, F_a(\Uparrow_C^w (v)))$$

Similarly, we have

$$\Downarrow_C^{wa} (v) = F_\epsilon(\overline{b}, F_a(\Downarrow_C^w (v)))$$

where $\overline{b}$ is the boolean negation of b.

The description of the algorithm is divided into two parts, a pre–processing phase and a main loop, shown in Figure 12.7 and Figure 12.8, respectively. Input to the algorithm is a constraint set C and variables α, β. The algorithm *accepts* if and only if the *non–entailment* $C \not\models \alpha \leq \beta$ holds over general (finite and infinite) trees ordered structurally. Correctness of the algorithm is founded on Theorem 12.3.2, Theorem 10.4.2, Lemma 10.2.4 and Theorem 10.2.3, as is argued below.

Pre–processing (Figure 12.7)

Figure 12.7 defines a pre–processing phase which performs a number of checks. Once this phase has been completed, the algorithm performs the operations described in Figure 12.8, unless the pre–processing phase already results in acceptance or rejection. The first part checks that C is satisfiable. This check is implemented by checking weak unifiability and ground consistency, and correctness of this follows from Theorem 10.4.2. Both steps can obviously be performed in polynomial time (hence also polynomial space). We assume that the closure $\mathsf{Cl}(C)$ is performed on the constraint graph $\mathcal{G}_C$. The second part assumes (by the previous part) that C is satisfiable and checks that α and β occur non–trivially in C. If not, the algorithm accepts (*i.e.*, non–entailment is accepted.) Correctness follows from Lemma 10.2.4. The second step of this check determines whether or not the shapes of α and β in C are the same. This check can be performed by computing $U_C^\star(\alpha)$ and $U_C^\star(\beta)$, where $U^\star$ is the unifier U_C with all variables collapsed to the same element (arbitrarily chosen to be $\star$). The third step determines whether it is the case that one but not both of $\Theta_C(\alpha)$ or $\Theta_C(\beta)$ has a leaf variable that is forced to be mapped to a constant; this check can be performed by inspecting whether $\star$ is present in one (but not both) of the equivalence classes of α and β with respect to the most general unifier U_C (equivalence classes of U_C are denoted $[\bullet]_C$).

Main loop (Figure 12.8)

The second phase of the algorithm can assume that C is satisfiable and that α and β occur non–trivially in C, by the pre–processing phase. The second phase, shown in Figure 12.8, non–deterministically checks non–entailment over C-shaped valuations. Theorem 12.3.2 together with Theorem 10.2.3 show that this phase is correct. The algorithm stores only a bit b and the sets $U\alpha$, $L\beta$ which are subsets of the set V of vertices in the constraint graph, hence it certainly consumes only polynomially bounded space.

We can now conclude from the PSPACE–algorithm together with Theorem 12.1.2 that we have

Theorem 12.4.1 *The problem of structural, recursive subtype entailment is PSPACE–complete.*

Given C, α, β, check that C is satisfiable (consistent). If C is inconsistent, reject.

1. (Check for weak unifiability) Compute U_C (most general unifier admitting infinite terms). If unification fails, reject.
2. (Check for ground consistency) Compute $\mathsf{Cl}(C)$. If $\mathsf{Cl}(C)$ is not ground consistent, reject.

Given consistent C, α and β. Check that α and β are non–trivially related in C. Non–entailment $C \not\models \alpha \leq \beta$ is accepted, if α and β are trivially related.

1. (Occurrence check) Check that both α and β occur in C; if not, then accept.
2. (Match check) Check that $U^\star_C(\alpha) = U^\star_C(\beta)$; if not, then accept;
3. (Constant constraint check) If $\star \in [\alpha]_C$ and $\star \notin [\beta]_C$, or $\star \notin [\alpha]_C$ and $\star \in [\beta]_C$, then accept

Figure 12.7: PSPACE algorithm for entailment. Preprocessing phase.

Given C, α and β with C consistent and α, β occurring non–trivially in C. The algorithm accepts if and only if $C \not\models \alpha \leq \beta$ over C–shaped valuations in infinite trees ordered structurally.

1. (Initialize for loop)
   $S := U_C(\alpha)$;
   $b := 0$;
   $U\alpha :=\Uparrow^{\epsilon}_{C} (\alpha)$;
   $L\beta :=\Downarrow^{\epsilon}_{C} (\beta)$;

2. (Loop)
   If $U\alpha \cap L\beta \neq \emptyset$ then reject.
   If $\mathsf{Con}(S) \in L \cup \mathcal{V}$ (leaf address reached) and $\bigwedge^b U\alpha \leq^b_L \bigvee^b L\beta$ then reject

3. Case S of:

   $S_1 \times S_2$: Guess $a := f$ or s;
   $U\alpha := F_U(b, U\alpha)$;
   $L\beta := F_L(b, L\beta)$;
   $S := S.a$; goto Step 2.

   $S_1 \to S_2$: Guess $a := d$ or r;
   If $a = d$ then $b := \overline{b}$;
   $U\alpha := F_U(b, U\alpha)$;
   $L\beta := F_L(b, L\beta)$;
   $S := S.a$; goto Step 2.

   B: Accept.

where :

$$F_U(b, U\alpha) = F_\epsilon(b, F_a(U\alpha))$$
$$F_L(b, L\beta) = F_\epsilon(\overline{b}, F_a(L\beta))$$

and $S.a$ is defined by

$$\begin{array}{lcll} S.f & = & S_1, & \text{if } S = S_1 \times S_2 \\ S.s & = & S_2, & \text{if } S = S_1 \times S_2 \\ S.d & = & S_1, & \text{if } S = S_1 \to S_2 \\ S.r & = & S_2, & \text{if } S = S_1 \to S_2 \end{array}$$

Figure 12.8: PSPACE algorithm for entailment. Main loop.

Chapter 13

Conclusion to Part II

13.1 Significance of the results

We have already given a table summarising the main results of this part of the thesis (see the Introduction, Chapter 7). In relation to the overall topic of this thesis, the main lesson to be drawn from this part of the thesis is that entailment and hence simplification in entailment based subtyping systems becomes intractable, as soon as we have non–flat structures of trees. The only source of complexity is syntactic structure, and already at the simplest level, syntactic structure takes us from a linear time problem (atomic entailment) to a coNP–complete problem (finite, structural entailment); non–structural order even takes us to a PSPACE–hard problem (finite, non–structural entailment). We have shown that, in the structural case, recursive types increase complexity, and we have shown that, in the finite case, non–structural order is more difficult that structural order. We conjecture (Conjecture 9.4.5) that the non–structural problems are in PSPACE hence PSPACE–complete. We have also shown that the satisfiability problem for structural recursive subtyping is in PTIME.

Comparing the table in Section 2.4.4, showing the complexity of the satisfiability problems, with the table summarising our results on the corresponding entailment complexity (see the Introduction, Chapter 7), we see two things. First, entailment complexity is much higher than the complexity of satisfiability, indicating that the problem of representing typings succinctly is much more difficult than solving the typability problem in subtyping systems. Second, it appears that the two tables are "isomorphic", in the sense that entailment complexity appears to be a blown–up version

of the table for satisfiability (with $M(n)$ corresponding to coNP and cubic time corresponding to PSPACE.) This would be the more striking, if our Conjecture 9.4.5 turns out to be correct.

13.2 Related work

Bloniarz, Hunt and Rosenkrantz [39] studied the complexity of deciding logical equivalence of constant–free functional expressions over lattices and a variety of other algebraic structures [11, 12]. Their work includes complexity results on several forms of minimization of such expressions, with motivation related to ours (*e.g.*, minimizing functional expressions over a lattice in order to speed up data flow analysis.) These problems are characterized by the presence of strong algebraic operations in the formal languages and the absence of syntactic structure. In comparison to this, the formalisms studied in the present thesis are characterized by the absence of algebraic operations (like formal meet and join) and the presence of syntactic structure (type expressions and trees).

Cosmadakis [15] studied the uniform word problem for lattices, *i.e.*, the problem of deciding $E \models \phi = \psi$, where ϕ, ψ are formal expressions built from variables, meet and join, and where E is a set of equations between such expressions. Entailment is uniform as in first order model theory, so in our jargon the problem is to decide whether $E \models_L \phi = \psi$ for all lattices L. Interestingly, the problem is in PTIME, by a complete axiomatization. Again, a major difference in comparison to the work presented in this thesis is that we are dealing with complexity arising from syntax, not algebraic operations, and entailment in the present work is not uniform.

The study of entailment by Flanagan and Felleisen [25] is related to the present study, although they work in a different model of complete infinite trees labeled with sets of constructors, and their notion of entailment is different from ours. Even though the exact relation between their work and ours remains unclear, we have to some extent been inspired by their methods. Thus, Flanagan [24] proves PSPACE-hardness of their entailment predicate by reduction from the NFA containment problem (see [30].) However, the proof relies on a complicated (and impressive), complete axiomatization of entailment, whereas our proof uses non–syntactic methods and a different reduction from a special form of the NFA universality problem (their reduction appears not to be directly applicable to our cases.) Their axiomatization leads to a complete algorithm for entailment, but since it

runs in exponential time and consumes exponential space they do not give a tight classification of complexity.

13.3 Open problems

The most obvious and pressing open problem is to settle Conjecture 9.4.5. Another interesting open problem is to determine decidability and complexity of the "semantic" relations, with the the logical form of $\prec_{sem}$, over various structures. The problem was conjectured to be undecidable over $\mathcal{T}_\Sigma[n]$ (non–structural recursive types) in [73].

Appendix A

Proofs

A number of proofs were left out of the main text. They are given in the following sections. Each section is devoted to proofs for a chapter in the main text. Proofs for theorems and lemmas should be read in the context of the main text in which they are stated, *i.e.*, notation, definitions etc. will not be redefined here.

A.1 Proofs for Chapter 2

Proof of Lemma 2.3.5

Lemma 2.3.5 *In the model $\mathcal{T}_\Sigma^{\mathrm{F}}[s]$ one has:*

1. *If $v \models C$, then $v(C)$ is matching.*
2. *If $v \models C$, then $v = v' \circ \Theta_C$ holds on the variables in C for some valuation v'*
3. *$C \models \tau \leq \tau'$ if and only if $\Theta_C(C) \models \Theta_C(\tau) \leq \Theta_C(\tau')$*
4. *If C is an atomic, satisfiable set and $C \models_P \tau \leq \tau'$, then τ and τ' are matching.*

PROOF The first claim is an immediate consequence of Lemma 2.1.1. The second claim is a consequence of the first together with the properties of Θ_C as most general matching substitution. To prove the third claim, suppose that $C \models \tau \leq \tau'$, and let $v \models \Theta_C(C)$. Then $v \circ \Theta_C \models C$, hence $v \circ \Theta_C \models \tau \leq \tau'$, hence $v \models \Theta_C(\tau) \leq \Theta_C(\tau')$, showing $\Theta_C(C) \models \Theta_C(\tau) \leq \Theta_C(\tau')$.

Conversely, assuming $\Theta_C(C) \models \Theta_C(\tau) \leq \Theta_C(\tau')$, let $v \models C$. Then, by the first part of the lemma, we have $v = v' \circ \Theta_C$ for some valuation v', and therefore we have $v' \models \Theta_C(C)$. By the assumption, it follows that $v' \models \Theta_C(\tau) \leq \Theta_C(\tau')$, *i.e.*, $v' \circ \Theta_C \models \tau \leq \tau'$, and hence $v \models \tau \leq \tau'$ (the variables in τ and τ' must occur in C, by $\Theta_C(C) \models \Theta_C(\tau) \leq \Theta_C(\tau')$, so $v' \circ \Theta_C = v$ on all relevant variables.)

To prove the last claim, suppose that $C \models_P \tau \leq \tau'$ with C atomic and satisfiable. Then there is a valuation v in P such that $v \models_P C$, hence $v(\tau) \leq v(\tau')$ holds in $\mathcal{T}^{\mathrm{F}}_{\Sigma}[s]$. Then $v(\tau)$ and $v(\tau')$ are matching, by Lemma 2.1.1. Since v is a valuation in P, τ and τ' must already be matching. □

Proof of Lemma 2.3.10

Lemma 2.3.10 *Let C be a weakly unifiable constraint set. Then*

1. *C is satisfiable if and only if $C^\flat$ is satisfiable. More specifically, one has*
   (a) *If $v \models C$, then there is a valuation v' such that $v = v' \circ \Theta_C$ holds on the variables in C, and with $v' \models C^\flat$*
   (b) *$v \models C^\flat$ if and only if $v \circ \Theta_C \models C$*
2. *If α and β are matching in C, then*

$$C \models \alpha \leq \beta \text{ if and only if } C^\flat \models [\alpha \leq \beta]_C$$

PROOF To prove the first claim, assume $v \models C$. Then, by the most general matching substitution property for Θ_C together with Lemma 2.3.5 we have

$$v = v' \circ \Theta_C \text{ for some } v' \tag{A.1}$$

Then $v' \models C^\flat$ follows from Lemma 2.3.8 applied to $\Theta_C(C)$.

To prove the third claim, assuming $v \models C^\flat$, we can show that

$$v \circ \Theta_C \models C \tag{A.2}$$

To see (*A*.2), let $\tau \leq \tau' \in C$, then $\Theta_C(\tau)$ and $\Theta_C(\tau')$ are matching, and therefore they have the same set of leaf addresses. If w is such a leaf address, then it follows from $v \models C^\flat$ that we have

$$v(\Theta_C(\tau))(w) \leq^w v(\Theta_C(\tau'))(w)$$

and by Lemma 2.3.8 this shows that $(A.2)$ is true. Conversely, assume $v \circ \Theta_C \models C$. Then $v \models \Theta_C(C)$, from which we get $v \models C^\flat$ by Lemma 2.3.8.

To prove the second claim, consider first the implication $(\Rightarrow)$. Assume $C \models \alpha \leq \beta$ and let $v \models C^\flat$. Then $(A.2)$ holds, by the proof of the first claim of the lemma. Since $C \models \alpha \leq \beta$, $(A.2)$ entails that $v \circ \Theta_C \models \alpha \leq \beta$, hence $v \models \Theta_C(\alpha) \leq \Theta_C(\beta)$, and by Lemma 2.3.8 this allows us to conclude that $v \models [\alpha \leq \beta]_C$ as desired.

To prove the implication $(\Leftarrow)$, assume $C^\flat \models [\alpha \leq \beta]_C$ and let $v \models C$. Then $v' \models C^\flat$ by the first claim of this lemma, where v' is given by $(A.1)$ and therefore one has $v' \models [\alpha \leq \beta]_C$. Since α and β are assumed to be matching in C, we have $\Theta_C(\alpha)$ and $\Theta_C(\beta)$ matching. It then follows from Lemma 2.3.8 that $v' \models \Theta_C(\alpha) \leq \Theta_C(\beta)$, hence $v' \circ \Theta_C \models \alpha \leq \beta$, *i.e.*, $v \models \alpha \leq \beta$, as desired. □

A.2 Proofs for Chapter 3

Proof of Lemma 3.2.3

Lemma 3.2.3 *Let C be satisfiable.*

1. *If $C \models_P \alpha \leq A$ with $\alpha \neq A$ and $\alpha \notin \mathsf{Var}(C)$, then, for some $b \in P$, one has $C \models_P A = b$ and $\models_P \alpha \leq b$. In other words, P must have a top element equivalent to A.*

2. *If $C \models_P A \leq \alpha$ with $\alpha \neq A$ and $\alpha \notin \mathsf{Var}(C)$, then, for some $b \in P$, one has $C \models_P A = b$ and $\models_P b \leq \alpha$. In other words, P must have a bottom element equivalent to A.*

PROOF We only prove the first claim, since the second one is similar. So suppose that $C \models_P \alpha \leq A$ with $\alpha \notin \mathsf{Var}(C)$. Since $\alpha \notin \mathsf{Var}(C)$ and $\alpha \neq A$, it is easy to see that we have

$$\forall b_0 \in P.\ C \models_P b_0 \leq A \tag{A.3}$$

Suppose that v_1 and v_2 were any two valuations satisfying C and with $v_1(A) \neq v_2(A)$; then either $v_1(A) \not\leq_P v_2(A)$ or else $v_2(A) \not\leq_P v_1(A)$. Assume w.l.o.g. that $v_1(A) \not\leq_P v_2(A)$. Then, by $(A.3)$ we have $C \models_P v_1(A) \leq A$. Since $v_2 \models C$, it follows that $v_1(A) \leq_P v_2(A)$, a contradiction. We must therefore conclude that, for all valuations v_1, v_2 satisfying C, one has

$v_1(A) = v_2(A)$. By satisfiability of C, choose a valuation v with $v \models_P C$. Let $b = v(A)$. Then we must have $C \models_P A = b$ by the previous argument, and, moreover, $C \models_P \alpha \leq b$, by $(A.3)$. □

Proof of Lemma 3.2.6

Lemma 3.2.6 *The relation* $\ll$ *is transitive: if* $\mathbf{t}_1 \ll_{S_1} \mathbf{t}_2$ *and* $\mathbf{t}_2 \ll_{S_2} \mathbf{t}_3$, *then* $\mathbf{t}_1 \ll_{S_1 \circ S_2} \mathbf{t}_3$.

PROOF Let $\mathbf{t}_i = C_i, \Gamma_i \vdash_P M : \tau_i$ for $i = 1, 2, 3$. Suppose that $\mathbf{t}_1 \ll_{S_1} \mathbf{t}_2$ and $\mathbf{t}_2 \ll_{S_2} \mathbf{t}_3$. Then we can show that $\mathbf{t}_1 \ll_{S_1 \circ S_2} \mathbf{t}_3$. Only the property

$$C_1 \subseteq \mathsf{Ker}(S_1 \circ S_2(C_3)) \tag{A.4}$$

is not obvious, so we show only that. By the assumptions, we have

$$C_1 \subseteq \mathsf{Ker}(S_1(C_2)) \text{ and } C_2 \subseteq \mathsf{Ker}(S_2(C_3))$$

We now show that, for any constraint set C and substitution S, one has

$$\mathsf{Ker}(S(\mathsf{Ker}(C))) \subseteq \mathsf{Ker}(S(C)) \tag{A.5}$$

So suppose that $\phi \in \mathsf{Ker}(S(\mathsf{Ker}(C)))$; then $\phi \notin \mathsf{Th}(P)$, $\mathsf{Var}(\phi) \subseteq \mathsf{Var}(S(\mathsf{Ker}(C)))$ and $S(\mathsf{Ker}(C)) \models_P \phi$. Since $C \sim_P \mathsf{Ker}(C)$, we have $S(C) \sim_P S(\mathsf{Ker}(C))$, hence $S(C) \models_P \phi$. Moreover, we have

$$\mathsf{Var}(S(\mathsf{Ker}(C))) = S(\mathsf{Var}(\mathsf{Ker}(C))) \subseteq S(\mathsf{Var}(C)) = \mathsf{Var}(S(C))$$

and therefore $\mathsf{Var}(\phi) \subseteq \mathsf{Var}(S(C))$. We have now shown $\phi \in \mathsf{Ker}(S(C))$, thereby proving $(A.5)$.

To prove $(A.4)$, we now reason as follows, using previous relations and $(A.5)$:

$$\begin{array}{ll}
C_2 \subseteq \mathsf{Ker}(S_2(C_3)) & \Rightarrow \\
S_1(C_2) \subseteq S_1(\mathsf{Ker}(S_2(C_3))) & \Rightarrow \\
\mathsf{Ker}(S_1(C_2)) \subseteq \mathsf{Ker}(S_1(\mathsf{Ker}(S_2(C_3)))) & \Rightarrow \\
\mathsf{Ker}(S_1(C_2)) \subseteq \mathsf{Ker}(S_1(S_2(C_3))) & \Rightarrow \\
C_1 \subseteq \mathsf{Ker}(S_1 \circ S_2(C_3)) &
\end{array}$$

□

Proof of Theorem 3.2.8

Theorem 3.2.8 *If* $\mathbf{t}_1 \ll_{S_2} \mathbf{t}_2$ *and* $\mathbf{t}_2 \ll_{S_1} \mathbf{t}_1$ *then* S_i *is a renaming on* $\mathbf{t}_i$, $i = 1, 2$.

PROOF Assuming $\mathbf{t}_1 \ll_{S_2} \mathbf{t}_2$ and $\mathbf{t}_2 \ll_{S_1} \mathbf{t}_1$, it follows that we have

(*i*) $C_1 \subseteq \mathsf{Ker}(S_2 \circ S_1(C_1))$

(*ii*) $\tau_1 = S_2 \circ S_1(\tau_1)$

(*iii*) $\mathcal{D}(\Gamma_1) = \mathcal{D}(\Gamma_2)$ and $\Gamma_1(x) = S_2 \circ S_1(\Gamma_1(x))$, for $x \in \mathcal{D}(\Gamma_1)$.

By our assumptions, (*i*) follows, because we have $\mathbf{t}_1 \ll_{S_2 \circ S_1} \mathbf{t}_1$, by transitivity (Lemma 3.2.6). We get (*ii*) from the assumptions which yield $\tau_1 = S_2(\tau_2)$ and $\tau_2 = S_1(\tau_1)$, hence $S_2(\tau_2) = S_2(S_1(\tau_1))$. The property (*iii*) follows the same way. Consider now the action of the map $S_2 \circ S_1$ on C_1. By (*i*) we have

$$\mathsf{Var}(C_1) \subseteq \mathsf{Var}(\mathsf{Ker}(S_2 \circ S_1(C_1))) \subseteq \mathsf{Var}(S_2 \circ S_1(C_1))$$

Since evidently $|\mathsf{Var}(S_2 \circ S_1(C_1))| \leq |\mathsf{Var}(C_1)|$, it follows that $\mathsf{Var}(C_1) = \mathsf{Var}(S_2 \circ S_1(C_1)) = S_2 \circ S_1(\mathsf{Var}(C_1))$. We can conclude that

(*a*) $S_2 \circ S_1$ is a renaming on $\mathsf{Var}(C_1)$ mapping $\mathsf{Var}(C_1)$ to itself.

Moreover, (*ii*) shows that

(*a*) $S_2 \circ S_1$ is a renaming (in fact, the identity) on $\mathsf{Var}(\tau_1)$ mapping $\mathsf{Var}(\tau_1)$ to itself

and (*iii*) shows that

(*a*) $S_2 \circ S_1$ is a renaming (in fact, the identity) on $\mathsf{Var}(\Gamma_1)$ mapping $\mathsf{Var}(\Gamma_1)$ to itself

From (*a*), (*b*) and (*c*) we can conclude that the image of $\mathsf{Var}(\mathbf{t}_1)$ under $S_2 \circ S_1$ is $\mathsf{Var}(\mathbf{t}_1)$ itself, that is

$$S_2 \circ S_1(\mathsf{Var}(\mathbf{t}_1)) = \mathsf{Var}(\mathbf{t}_1)$$

Since $\mathsf{Var}(\mathbf{t}_1)$ is a finite set, we can conclude that $S_2 \circ S_1$ is a renaming on the entire set $\mathsf{Var}(\mathbf{t}_1)$. It then follows that S_1 must be a renaming on $\mathsf{Var}(\mathbf{t}_1)$. One proves that S_2 is a renaming on $\mathbf{t}_2$ by an entirely symmetric argument. □

Proof of Lemma 3.3.3

Lemma 3.3.3 *If* $\mathbf{t}_1 \prec_{S_1} \mathbf{t}_2$ *and* $\mathbf{t}_2 \prec_{S_2} \mathbf{t}_1$, *then* $S_1(\mathbf{t}_1) \prec_{S_2} \mathbf{t}_1$ *and* $S_2(\mathbf{t}_2) \prec_{S_1} \mathbf{t}_2$.

PROOF Let $\mathbf{t}_1 = C_1, \Gamma_1 \vdash_P M : \tau_1$, $\mathbf{t}_2 = C_2, \Gamma_2 \vdash_P M : \tau_2$. By the assumption, we have

(*i*) $C_2 \models_P S_1(C_1)$

(*ii*) $C_2 \models_P S_1(\tau_1) \leq \tau_2$

(*iii*) $\mathcal{D}(\Gamma_1) \subseteq \mathcal{D}(\Gamma_2)$ and $\forall x \in \mathcal{D}(\Gamma_1).\ C_2 \models_P \Gamma_2(x) \leq S_1(\Gamma_1(x))$

and

(*iv*) $C_1 \models_P S_2(C_2)$

(*v*) $C_1 \models_P S_2(\tau_2) \leq \tau_1$

(*vi*) $\mathcal{D}(\Gamma_2) \subseteq \mathcal{D}(\Gamma_1)$ and $\forall x \in \mathcal{D}(\Gamma_2).\ C_1 \models_P \Gamma_1(x) \leq S_2(\Gamma_2(x))$

By (*i*) and the Substitution Lemma (Lemma 2.3.14) we have

$$S_2(C_2) \models_P S_2(S_1(C_1)) \tag{A.6}$$

and so, by (*iv*) and (*A*.6), we get

$$C_1 \models_P S_2(S_1(C_1)) \tag{A.7}$$

Moreover, by (*ii*) and the Substitution Lemma we have

$$S_2(C_2) \models_P S_2(S_1(\tau_1)) \leq S_2(\tau_2) \tag{A.8}$$

hence, by (*iv*) and (*A*.8) we get

$$C_1 \models_P S_2(S_1(\tau_1)) \leq S_2(\tau_2) \tag{A.9}$$

and then, by (*v*) and (*A*.9)

$$C_1 \models_P S_2(S_1(\tau_1)) \leq \tau_1 \tag{A.10}$$

In analogous manner one obtains

$$C_1 \models_P \Gamma_1(x) \leq S_2(S_1(\Gamma_1(x))) \tag{A.11}$$

Then (*A*.7), (*A*.10) and (*A*.11) show that $S_1(\mathbf{t}_1) \prec_{S_2} \mathbf{t}_1$. The proof that $S_2(\mathbf{t}_2) \prec_{S_1} \mathbf{t}_2$ is symmetric and left out. □

Proof of Lemma 3.4.2

Lemma 3.4.2 *Let* $\mathbf{t} = C, \Gamma \vdash_P M : \tau$ *be any atomic judgement.*

1. *If* $C \models_P \alpha = A$, *then* $S(\mathbf{t}) \approx \mathbf{t}$, *with* $S = \{\alpha \mapsto A\}$.
2. *There is a substitution instance* $\mathbf{t}'$ *of* $\mathbf{t}$ *such that* $\mathbf{t}'$ *has an acyclic constraint set and with* $\mathbf{t}' \approx \mathbf{t}$.
3. *If* $\mathbf{t}$ *is fully substituted with* C *satisfiable, then* C *is acyclic.*

PROOF To show that $S(\mathbf{t}) \approx \mathbf{t}$ it is sufficient to show $S(\mathbf{t}) \prec \mathbf{t}$, and it is easy to verify that one has $S(\mathbf{t}) \prec_{id} \mathbf{t}$, whenever the conditions of the lemma are satisfied.

By repeated application of non–renaming substitutions S of the form mentioned in the lemma, one evidently obtains a typing with acyclic constraint set. This shows that the second claim of the lemma is true.

By the previous observations, any cycle of the form $C \models \alpha = A$ with $A \neq \alpha$ gives rise to a non–renaming substitution $S = \{\alpha \mapsto A\}$ such that $S(\mathbf{t}) \in [\mathbf{t}]$; this shows that no typing with cyclic constraint set can be fully substituted. □

Proof of Lemma 3.4.4

Lemma 3.4.4 *Let* C_1 *and* C_2 *be atomic constraint sets.*

1. *If* $\mathsf{Ker}(C_1) \subseteq \mathsf{Ker}(C_2)$ *then* $C_2 \models_P C_1$
2. *If* C_1 *is acyclic and satisfiable, then* $C_1 \models_P C_2$ *implies* $\mathsf{Ker}(C_2) \subseteq \mathsf{Ker}(C_1)$
3. *If* C_1 *and* C_2 *are both acyclic and satisfiable, then* $C_1 \sim_P C_2$ *if and only if* $\mathsf{Ker}(C_1) = \mathsf{Ker}(C_2)$

PROOF To see the first claim, assume that $\mathsf{Ker}(C_1) \subseteq \mathsf{Ker}(C_2)$, and let $\phi \in C_1$. If $\phi \in \mathsf{Ker}(C_1)$, then $C_2 \models_P \phi$ follows immediately from the assumptions. If $\phi \notin \mathsf{Ker}(C_1)$, then $\phi \in \mathsf{Th}(P)$ (because $\mathsf{Var}(\phi) \subseteq \mathsf{Var}(C_1)$), and so $C_2 \models_P \phi$.

To prove the second claim, assume that C_1 is acyclic with $C_1 \models_P C_2$, and suppose that $\phi \in \mathsf{Ker}(C_2)$. Then we have $C_2 \models_P \phi$, $\mathsf{Var}(\phi) \subseteq \mathsf{Var}(C_2)$ and $\phi \notin \mathsf{Th}(P)$. We have $C_1 \models_P \phi$, because $C_1 \models_P C_2$ and $\models_P$ is transitive. We

only need to show, then, that $\mathsf{Var}(\phi) \subseteq \mathsf{Var}(C_1)$. To see this, suppose for the sake of contradiction that $\mathsf{Var}(\phi) \not\subseteq \mathsf{Var}(C_1)$. Then we can write $\phi = \alpha \leq A$ with $\alpha \notin \mathsf{Var}(C_1)$ (the case $\phi = A \leq \alpha$ with $\alpha \notin \mathsf{Var}(C_1)$ is similar and left out.) Since $\phi \notin \mathsf{Th}(P)$, we must have $\alpha \neq A$. By Lemma 3.2.3 we then have $C_1 \models_P A = b$ with $\models_P \alpha \leq b$ for some $b \in P$. If $A = b$, then $\phi = \alpha \leq b$ and so $\phi \in \mathsf{Th}(P)$, which is not the case, so we must assume $A \neq b$. But $A \neq b$ together with $C_1 \models_P A = b$ implies that C is cyclic, which contradicts our assumptions. We must conclude that $\mathsf{Var}(\phi) \subseteq \mathsf{Var}(C_1)$.

The third claim follows from the second. □

Proof of Lemma 3.4.5

We first need the following lemma:

Lemma A.2.1 *Let S be a renaming on C. Then $\mathsf{Ker}(S(C)) = S(\mathsf{Ker}(C))$.*

PROOF To see that $S(\mathsf{Ker}(C)) \subseteq \mathsf{Ker}(S(C))$, suppose that $\phi \in S(\mathsf{Ker}(C))$. Then there is an inequality $\psi \in \mathsf{Ker}(C)$ such that $C \models_P \psi$ and $\phi = S(\psi)$. Since $\psi \in \mathsf{Ker}(C)$, we have $\psi \notin \mathsf{Th}(P)$. Because S is a renaming, it then follows easily that $\phi \notin \mathsf{Th}(P)$. By $C \models_P \psi$, we get $S(C) \models_P \phi$, by substitutivity. Now we only need to show that $\mathsf{Var}(\phi) \subseteq \mathsf{Var}(S(C))$. Observe first that we have $\mathsf{Var}(\mathsf{Ker}(C)) \subseteq \mathsf{Var}(C)$, hence $S(\mathsf{Var}(\mathsf{Ker}(C))) \subseteq S(\mathsf{Var}(C))$; but $S(\mathsf{Var}(C)) = \mathsf{Var}(S(C))$. We can conclude that $\mathsf{Var}(S(\mathsf{Ker}(C))) \subseteq \mathsf{Var}(S(C))$. Since $\phi \in S(\mathsf{Ker}(C))$, it then follows that $\mathsf{Var}(\phi) \subseteq \mathsf{Var}(S(C))$.

To see that $\mathsf{Ker}(S(C)) \subseteq S(\mathsf{Ker}(C))$, suppose that $\phi \in \mathsf{Ker}(S(C))$, so that $S(C) \models_P \phi$ with $\mathsf{Var}(\phi) \subseteq \mathsf{Var}(S(C))$ and $\phi \notin \mathsf{Th}(P)$. Since S is a renaming on $\mathsf{Var}(C)$, it has an inverse S^{-1}, which is a bijection of $S(\mathsf{Var}(C))$ onto $\mathsf{Var}(C)$. By $S(C) \models_P \phi$, we then get $C \models_P S^{-1}(\phi)$. It is sufficient to show $S^{-1}(\phi) \in \mathsf{Ker}(C)$. Since $\mathsf{Var}(\phi) \subseteq \mathsf{Var}(S(C)) = S(\mathsf{Var}(C))$, we have $\mathsf{Var}(S^{-1}(\phi)) = S^{-1}(\mathsf{Var}(\phi)) \subseteq \mathsf{Var}(C)$. Since S^{-1} is a renaming on $\mathsf{Var}(\phi)$, we get $S^{-1}(\phi) \notin \mathsf{Th}(P)$ from $\phi \notin \mathsf{Th}(P)$. This shows that $S^{-1}(\phi) \in \mathsf{Ker}(C)$, thereby proving the desired inclusion. □

Lemma 3.4.5 *Let C_1, C_2 be atomic constraint sets, both of which are acyclic and satisfiable. Assume that $C_1 \models_P S_2(C_2)$ and $C_2 \models_P S_1(C_1)$ where S_i is*

a renaming on $\mathsf{Var}(C_i)$, $i = 1, 2$. *Then* $C_1 \sim_P S_2(C_2)$ *and* $C_2 \sim_P S_1(C_1)$.

PROOF Assume $C_1 \models_P S_2(C_2)$ and $C_2 \models_P S_1(C_1)$ where S_i is a renaming on $\mathsf{Var}(C_i)$. Since S_i is a renaming on C_i with C_i acyclic and satisfiable, it easily follows that $S_i(C_i)$ is again acyclic and satisfiable, $i = 1, 2$. By Lemma A.2.1 and Lemma 3.4.4, it is then sufficient to prove $(A.12)$ and $(A.13)$:

$$\mathsf{Ker}(C_1) = S_2(\mathsf{Ker}(C_2)) \tag{A.12}$$

$$\mathsf{Ker}(C_2) = S_1(\mathsf{Ker}(C_1)) \tag{A.13}$$

Now, $C_1 \models_P S_2(C_2)$ and $C_2 \models_P S_1(C_1)$ imply (by Lemma 3.4.4) that we have $\mathsf{Ker}(S_2(C_2)) \subseteq \mathsf{Ker}(C_1)$ and $\mathsf{Ker}(S_1(C_1)) \subseteq \mathsf{Ker}(C_2)$, hence (by Lemma A.2.1)

$$S_2(\mathsf{Ker}(C_2)) \subseteq \mathsf{Ker}(C_1) \tag{A.14}$$

and

$$S_1(\mathsf{Ker}(C_1)) \subseteq \mathsf{Ker}(C_2) \tag{A.15}$$

Because S_i is a renaming on C_i, it must be the case that S_i is a renaming on $\mathsf{Ker}(C_i)$ $(i = 1, 2)$; S_i is therefore an injection on $\mathsf{Ker}(C_i)$ $(i = 1, 2)$. Because $\mathsf{Ker}(C_i)$ is a finite set for $i = 1, 2$, it then follows from $(A.14)$ and $(A.15)$ that we have

$$|\mathsf{Ker}(C_2)| = |S_2(\mathsf{Ker}(C_2))| \leq |\mathsf{Ker}(C_1)| = |S_1(\mathsf{Ker}(C_1))| \leq |\mathsf{Ker}(C_2)|$$

These relations show that

$$|S_2(\mathsf{Ker}(C_2))| = |\mathsf{Ker}(C_1)| \text{ and } |S_1(\mathsf{Ker}(C_1))| = |\mathsf{Ker}(C_2)|$$

Consequently, $(A.14)$ and $(A.15)$ show that $(A.12)$ and $(A.13)$ must be true. □

Proof of Lemma 3.4.6

Lemma 3.4.6 *Let* C *be atomic and satisfiable. If* C *is acyclic, then* $C \models_P \tau \leq \tau'$ *and* $C \models_P \tau' \leq \tau$ *imply* $\tau = \tau'$.

PROOF Assuming $C \models_P \tau \leq \tau'$ and $C \models_P \tau' \leq \tau$, we have $C \models_P \tau = \tau'$. We prove $\tau = \tau'$ by induction on the size of τ. Assume first that τ is an atom. Then, by the assumptions of the lemma together with the Match Lemma (Lemma 2.3.5), it follows that τ' is also an atom. But $C \models_P A = A'$ implies $A = A'$, because C is acyclic and satisfiable. Assume, for the inductive step,

that $\tau = \tau_1 \to \tau_2$. Then the Match Lemma implies that $\tau' = \tau_1' \to \tau_2'$, and $C \models_P \tau = \tau'$ then implies (via Decomposition, Lemma 2.3.13) that we have $C \models_P \{\tau_1 = \tau_1', \tau_2 = \tau_2'\}$. The claim now follows ny induction hypothesis applied to this entailment. Remaining cases are similar and left out. □

Proof of Lemma 3.5.1

The proof of this lemma has a simple abstract core, which it may be instructive to sketch before giving the proof in detail. Suppose $f : A \to A$ is a bijection of a finite set A onto itself, and suppose that $\leq$ is any binary relation on A (we use a very suggestive notation, $\leq$, but it could be any binary relation) such that f is "monotone" with respect to $\leq$, *i.e.*, $a \leq a' \Rightarrow f(a) \leq f(a')$. Then it will be the case, for any $a \in A$, that if $f(a) \neq a$ and either $f(a) \leq a$ or $a \leq f(a)$, then A must contain a proper cycle with respect to $\leq$, *i.e.*, there is a sequence of elements $a_1, \ldots, a_n$ in A with $a_1 < \ldots < a_n$ and $a_1 = a_n$, where $a < a'$ means that $a \leq a'$ and $a \neq a'$. To see this, consider that f injective and $f(a) \neq a$ imply $f^n(a) \neq f^{n-1}(a)$ for all n, and, moreover, assuming $f(a) \leq a$ (the other case where $a \leq f(a)$ is similar) we also have $f^n(a) \leq f^{n-1}(a)$ for all n, by "monotonicity" of f; in total we have $f^n(a) < f^{n-1}(a)$ for all n. Now consider the set $V = \{f^n(a) \mid n \geq 0\}$. Since $V \subseteq A$ is finite, there must be $i < j$ with $f^j(a) = f^i(a)$. Then $f^j(a) < f^{j-1}(a) < \ldots < f^i(a)$ constitutes a proper cycle in A.

Lemma 3.5.1 *Let S be a substitution and C an atomic, satisfiable constraint set with variable $\alpha \in \mathsf{Var}(C)$. Assume*

(*i*) *S is a renaming on $\mathsf{Var}(C)$*

(*ii*) *$C \models_P S(C)$*

(*iii*) *C is acyclic*

(*iv*) *either $C \models_P S(\alpha) \leq \alpha$ or $C \models_P \alpha \leq S(\alpha)$*

Then $S(\alpha) = \alpha$.

PROOF The proof is by contradiction, assuming (*i*), (*ii*), (*iii*), (*iv*) and $S(\alpha) \neq \alpha$. We show how to derive a contradiction under the assumption

that $C \models_P S(\alpha) \leq \alpha$; the proof under the alternative assumption $C \models_P \alpha \leq S(\alpha)$ is similar and left out.

So assume $\alpha \in \mathsf{Var}(C)$, $S(\alpha) \neq \alpha$, (i), (ii), (iii) and $C \models_P S(\alpha) \leq \alpha$. We will derive a contradiction. We first show the following *claim*:

For all $n > 0$ one has:

(a) *The set* $D_n = \{S^k(\alpha) \mid 0 \leq k \leq n\}$ *satisfies*

$$D_n \subseteq \mathsf{Var}(C)$$

(b) $S^n(\alpha) \neq S^{n-1}(\alpha)$

(c) $C \vdash_P S^n(\alpha) \leq S^{n-1}(\alpha)$

All three items are proven simultaneously by induction on $n > 0$. For $n = 1$, the set D_1 is just the set $\{\alpha, S(\alpha)\}$, so to establish (a) we need only show that $S(\alpha) \in \mathsf{Var}(C)$. By assumption we have $C \models_P S(\alpha) \leq \alpha$ and $S(\alpha) \neq \alpha$. Since S is a renaming on C, we have that $S(\alpha)$ is a variable, which is distinct from α. By our remaining assumptions (C acyclic, satisfiable) Lemma 3.4.3 applies, and it shows that $S(\alpha) \in \mathsf{Var}(C)$. To see (b) for $n = 1$, we need only show that $S(\alpha) \neq \alpha$, which holds by assumption. To see (c) for $n = 1$, we need only show that $C \models_P S(\alpha) \leq \alpha$, which holds by assumption. We have now shown the *claim* in the base case where $n = 1$.

For the inductive step, assume $n > 1$. We have by induction hypothesis that the set $D_{n-1} \subseteq \mathsf{Var}(C)$, hence, in particular, $S^{n-1}(\alpha), S^{n-2}(\alpha) \in \mathsf{Var}(C)$. By induction hypothesis applied to (c), we have $C \models_P S^{n-1}(\alpha) \leq S^{n-2}(\alpha)$, hence we get by the Substitution Lemma that $S(C) \models S^n(\alpha) \leq S^{n-1}(\alpha)$ and so, by (ii), $C \models_P S^n(\alpha) \leq S^{n-1}(\alpha)$. This shows that (c) holds in the inductive case. Further, by induction hypothesis applied to (b), we have $S^{n-1}(\alpha) \neq S^{n-2}(\alpha)$, and so, since S is injective as renaming on $\mathsf{Var}(C)$ with $S^{n-1}(\alpha), S^{n-2}(\alpha) \in \mathsf{Var}(C)$, we have $S^n(\alpha) \neq S^{n-1}(\alpha)$, which establishes (b) for the inductive case. Now, to see (a), note that, since $S^n(\alpha)$ and $S^{n-1}(\alpha)$ are two distinct variables (by (b) as just shown) which are comparable under the entailment hypotheses in C (by (c) as just shown), it follows from Lemma 3.4.3 that both variables must be mentioned in C, hence, in particular, $S^n(\alpha) \in \mathsf{Var}(C)$. Since we have, by induction hypothesis, that $D_{n-1} \subseteq \mathsf{Var}(C)$, it follows that $D_n \subseteq \mathsf{Var}(C)$. The proof of the *claim* is now complete.

Now, to prove the lemma, consider the set

$$V = \{S^n(\alpha) \mid n \geq 0\}$$

By property (a) of our *claim* above, we have $V \subseteq \mathsf{Var}(C)$, and therefore V is a finite set of variables with S an injection of V into itself (and, by finiteness of V, S is therefore a bijection of V onto itself.) Hence, by finiteness of V, there must exist $i < j$ such that $S^j(\alpha) = S^i(\alpha)$. By property (b) of the *claim* we have $S^j(\alpha) \neq S^{j-1}(\alpha)$, and by property (c) one easily sees that $C \models_P S^j(\alpha) \leq S^i(\alpha)$ whenever $i \leq j$. We have therefore established that C entails the relations

$$S^j(\alpha) < S^{j-1}(\alpha) < \ldots < S^i(\alpha)$$

with $S^i(\alpha) = S^j(\alpha)$ (and $S^j(\alpha) < S^{j-1}(\alpha)$ shorthand for $C \models_P S^j(\alpha) \leq S^{j-1}(\alpha)$ with $S^j(\alpha) \neq S^{j-1}(\alpha)$.) The sequence shown establishes that C is cyclic. We have now obtained the desired contradiction, because we have assumed (iii), that C is acyclic. □

A.3 Proofs for Chapter 5

Proof of Lemma 5.3.1

To prepare for a proof that the $\mathbf{Q}_n$ behave as claimed in the lemma, let α and β be two fixed, distinct variables and let $\upsilon_0, \upsilon_1, \ldots, \upsilon_k, \ldots$ and $\omega_0, \omega_1, \ldots, \omega_k, \ldots$ be two distinct enumerations of infinitely many type variables (so, all the ω_i are different from all the υ_j and all of these are distinct from α and β.) Let T_n denote the full binary tree of height n with 2^n leaf nodes, with internal nodes labeled by $\times$ and leaf nodes labeled by variables

$$\upsilon_0, \upsilon_{m+1}, \ldots, \upsilon_{2^n-1}$$

from left to right. So, for instance, T_0 is just the variable υ_0, T_1 is the type $\upsilon_0 \times \upsilon_1$, T_2 is the type $(\upsilon_0 \times \upsilon_1) \times (\upsilon_2 \times \upsilon_3)$, and so on. Let us say that a type τ *has the shape of* T_n if τ is built from $\times$ and variables only and, moreover, τ matches T_n; so, in this case, τ differs from T_n only by having possibly other variables than T_n at the leaves.

Define the type $\tau^{[n]}$ for $n \geq 0$ by setting

$$\begin{array}{lcl} \tau^{[0]} & = & \alpha \to (\beta \to T_1), \\ \tau^{[n]}(n > 0) & = & (\sigma_n \to \omega_n) \to \ldots (\sigma_1 \to \omega_1) \to \\ & & \alpha \to \beta \to T_{n+1} \end{array}$$

where σ_i is a renaming of T_i for $i = 1 \ldots n$, using fresh variables which occur nowhere else in the type.

Lemma 5.3.1 *Let P be any non–trivial poset. For all $n \geq 0$, there is a minimal principal typing for $\mathbf{Q}_n$ having the form $C_n, \emptyset \vdash_P \mathbf{Q}_n : \tau^{[n]}$, where all variables in C_n are observable, and C_n contains at least 2^{n+1} distinct variables.*

PROOF We prove the following property by induction on $n \geq 0$:

(∗) There is a minimal principal typing of $\mathbf{Q}_n$ having the form $C_n, \emptyset \vdash_P \mathbf{Q}_n : \tau^{[n]}$, where all variables in C_n are observable, and C_n contains at le ast 2^{n+1} distinct variables occurring in in T_{n+1}, constituting 2–crowns with α and β at the bottom, where α and β occur negatively in $\tau^{[n]}$ and the remainder of the variables occur positively in $\tau^{[n]}$.

To prove (∗), consider the term $\mathbf{Q}_0$ for the base case. We have already seen that a principal typing for $\mathbf{Q}_0$ has the form

$$C_0, \emptyset \vdash_P \lambda x : \alpha.\lambda y : \beta.cond_{x,y} : \alpha \to (\beta \to \upsilon_0 \times \upsilon_1)$$

with $C_0 = \{\alpha \leq \upsilon_0, \alpha \leq \upsilon_1, \beta \leq \upsilon_0, \beta \leq \upsilon_1\}$. This typing is derived by the *standard procedure* described earlier, where G-simplification has been performed to eliminate internal variables. All variables in C_0 are observable, and, moreover, C_0 is a 2-crown, with variables α and β occurring negatively, the remainder of the variables occrring positively, and α and β are at the bottom of the crown. It follows from Lemma 4.2.2 that the typing is S-simplified. It then follows from Theorem 4.2.5 that the typing is a *minimal* principal typing for $\mathbf{Q}_0$, and this typing satisfies the claim of the lemma for the case $n = 0$.

For the inductive case, consider $\mathbf{Q}_{n+1} =$

$$\begin{array}{l}\lambda f_{n+1}.\lambda f_{[n]}.\lambda x.\lambda y.(\lambda z.\mathbf{K} \\ \quad (\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle \mathtt{z}, \langle \mathtt{2nd\ z}, \mathtt{1st\ z}\rangle\rangle \\ \quad \mathit{else}\ \langle\langle \mathtt{2nd\ z}, \mathtt{1st\ z}\rangle, \mathtt{z}\rangle) \\ \quad (f_{n+1}z)) \\ \quad (\mathbf{P}^n cond_{x,y})\end{array}$$

By induction hypothesis for $\mathbf{Q}_n$, $(\mathbf{P}^n cond_{x,y})$ has a minimal principal typing with type T_{n+1} and coercion set C_n with at least 2^{n+1} distinct variables,

assuming the appropriate types $\sigma_i \to \omega_i$ for the f_i $(i = 1 \ldots n)$, α for x and β for y. Now imagine that we have inserted coercions at the leaves in the remaining part of $\mathbf{Q}_{n+1}$ (cf. the *standard procedure.*) Consider the type which must be assumed for f_{n+1}. Because $(\mathbf{P}^n cond_{x,y})$ is applied to $\lambda z \ldots$, the type of z must be T_{n+1}, and due to the application $(f_{n+1} z)$, the type assumed for f_{n+1} must therefore have the form $\tau_1 \times \tau_2 \to \eta$, where $\tau_1 \times \tau_2$ is a renaming of T_{n+1} (using fresh variables, and η fresh.) However, since every variable in $\tau_1 \times \tau_2$ occurs only positively in the type of the entire expression $\mathbf{Q}_{n+1}$, it follows by S-simplification that $\tau_1 \times \tau_2$ can be identified with T_{n+1}, and hence we need apply no coercion to z or f_{n+1}. Using elimination of internal variables by G-simplification, it can be seen that a principal completion of $\mathbf{Q}_{n+1}$ is obtained by inserting coercions as follows (assuming suitable coercions inside $(\mathbf{P}^n cond_{x,y})$)

$$
\begin{array}{l}
\lambda f_{n+1} : T_{n+1} \to \eta . \lambda f_{[n]} : [\sigma'_n] . \lambda x : \alpha . \lambda y : \beta . \\
(\lambda z : T_{n+1} . \mathbf{K} \\
\quad (\mathit{if}\ \mathtt{true}\ \mathit{then} \\
\quad \langle \uparrow^{\theta_1 \times \theta_2}_{T_1 \times T_2} z, \langle \mathtt{2nd}\ \uparrow^{\mathtt{T}_1 \times \theta_3}_{\mathtt{T}_1 \times \mathtt{T}_2} \mathtt{z}, \mathtt{1st}\ \uparrow^{\theta_4 \times \mathtt{T}_2}_{\mathtt{T}_1 \times \mathtt{T}_2} \mathtt{z} \rangle \rangle \\
\quad \mathit{else} \\
\quad \langle \langle \mathtt{2nd}\ \uparrow^{\mathtt{T}_1 \times \theta_1}_{\mathtt{T}_1 \times \mathtt{T}_2} \mathtt{z}, \mathtt{1st}\ \uparrow^{\theta_2 \times \mathtt{T}_2}_{\mathtt{T}_1 \times \mathtt{T}_2} \mathtt{z} \rangle, \uparrow^{\theta_3 \times \theta_4}_{\mathtt{T}_1 \times \mathtt{T}_2} \mathtt{z} \rangle) \\
\quad (f_{n+1}\ z)) \\
\quad (\mathbf{P}^n cond_{x,y})
\end{array}
$$

Here T_1 and T_2 are such that $T_1 \times T_2 = T_{n+1}$, so T_1 and T_2 match each other and each has 2^n distinct variables, and $\lambda f_{[n]} : [\sigma'_n]$ abbreviates the list of typed parameters $\lambda f_i : \sigma_i \to \omega_i$, $i = 1 \ldots n$. The types θ_i are renamings, using fresh variables, of T_1 (or, equivalently, of T_2.) This shows that a principal typing for $\mathbf{Q}_{n+1}$ can be obtained by adding to C_n the two 2-crowns C_1, C_2 of coercions shown in the completion above:

$$
\begin{array}{rcl}
C_1 & = & \{T_1 \le \theta_1, T_1 \le \theta_2, T_2 \le \theta_2, T_2 \le \theta_1\} \\
C_2 & = & \{T_1 \le \theta_4, T_1 \le \theta_3, T_2 \le \theta_3, T_2 \le \theta_4\}
\end{array}
$$

Now, these crowns are not atomic, since the types in them are all of the shape of T_n, hence to add them to C_n we need to decompose them. It is easy to see that each C_i decomposes into a set C'_i of 2^n atomic 2-crowns (since T_n has 2^n leaves) resulting in a total of $2 \cdot 2^n = 2^{n+1}$ new crowns; since the θ_i have fresh variables, each C_i contains $2 \cdot 2^n = 2^{n+1}$ new, distinct variables, and since, by induction, C_n already has at least 2^{n+1} distinct variables, it follows that the number of distinct variables in $C_{n+1} = C_n \cup C'_1 \cup C'_2$ is at

least $2^{n+1} + 2^{n+1} = 2^{n+2}$. Moreover, since T_1 and T_2 contain 2–crowns with α and β at the bottom, it follows from the form of C_1 and C_2 that the new crowns have α and β at the bottom, too.

The type of $\mathbf{Q}_{n+1}$ under the principal typing shown has the form

$$(T_{n+1} \to \eta) \to \sigma'_n \to \ldots \to \sigma'_1 \to \\ \alpha \to \beta \to ((\theta_1 \times \theta_2) \times (\theta_3 \times \theta_4))$$

(with $\sigma'_i = \sigma_i \to \omega_i$) which can be renamed to $\tau^{[n+1]}$. Since, by induction, all variables in C_n are observable in $\tau^{[n]}$, and since all new variables in C_{n+1} are in the θ_i, it follows that all variables in C_{n+1} are observable in the typing of $\mathbf{Q}_{n+1}$. It is immediate from the type that α and β occur negatively, and the remaining variables occurring in T_{n+1}, σ_i and θ_i all occur positively in the type.

It remains to show that the typing shown for $\mathbf{Q}_{n+1}$ is *minimal*. By induction hypothesis, C_n is minimal as part of a minimal typing of $\mathbf{Q}_n$, hence, in particular, it is S-simplified. By the shape of 2-crowns and the structure of the type, Lemma 4.2.2 shows that adding the crowns of C'_1 and C'_2 preserves this property, so C_{n+1} is S-simplified also. It then follows from Theorem 4.2.5 that the typing shown for $\mathbf{Q}_{n+1}$ is minimal. This completes the proof of the lemma. □

Lower bound construction in pure λ–calculus

We show that the lower bound construction can be accomplished in pure λ–calculus, without assuming conditionals, pairing and projection.

First consider the conditional. The only property of the conditional used in the construction is that it requires the types of both of its branches, say M_1 and M_2, to be coerced to a common supertype. This can be effected without the conditional by placing M_1 and M_2 in the context

$$\lambda x.\mathbf{K}(x\ M_1)(x\ M_2)$$

which requires the types of M_1 and M_2 to be coerced to the domain type of x. This eliminates the need for the conditional.

Eliminating pairing and projections is more subtle. A first attempt might be to use the standard lambda-calculus encodings (see [7, Chapter 6.2]), taking an encoded pair of M and N to be $\langle\!|\, M, N \,|\!\rangle$ with the definition

$$\langle\!|\, M, N \,|\!\rangle = \lambda p.(p\ M)\ N$$

and $\pi_1 = \lambda x.\lambda y.x$, $\pi_2 = \lambda x.\lambda y.y$. This will not work, however, because the application of a variable z of encoded pair-type $(\tau \to \sigma \to \theta) \to \theta$ (corresponding to $\tau \times \sigma$) to both projections will force $\tau = \sigma$; this is well-known in simple types and the same identification is also made (via valid simplifications) in subtyping. [1] For example, the expression

$$M = (\lambda z.\langle\!|\, z\ (\lambda x.\lambda y.x), z\ (\lambda x.\lambda y.y)\,|\!\rangle)\ \langle\!|\, x, y\,|\!\rangle$$

gets principal typing

$$\emptyset, \{x : \alpha, y : \alpha\} \vdash_P M : (\alpha \to \alpha \to \beta) \to \beta$$

This identification of types will render the lower-bound proof invalid for these encodings. However, instead of writing expressions such as

$$\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle z, \langle \pi_2\ z, \pi_1\ z\rangle\rangle\ \mathit{else}\ \langle\langle \pi_2\ z, \pi_1\ z\rangle, z\rangle$$

we can do without the projections, by using the term

$$\mathit{if}\ \mathtt{true}\ \mathit{then}\ \langle\!|\, \langle\!|\, z, s\,|\!\rangle, \langle\!|\, z, t\,|\!\rangle\,|\!\rangle\ \mathit{else}\ \langle\!|\, \langle\!|\, s, z\,|\!\rangle, \langle\!|\, t, z\,|\!\rangle\,|\!\rangle$$

where s and t are distinct, free variables. Summing up, we use the definitions

$$\begin{aligned}
\mathit{eqty}\,(M, N) &= \lambda x.\mathbf{K}(xM)(xN) \\
\langle\!|\, M, N\,|\!\rangle &= \lambda p.(pM)N \\
\mathbf{P}_{f,s,t} &= \lambda z.\mathbf{K}\ \mathit{eqty}\,(\langle\!|\, \langle\!|\, z, s\,|\!\rangle, \langle\!|\, z, t\,|\!\rangle\,|\!\rangle, \langle\!|\, \langle\!|\, s, z\,|\!\rangle, \langle\!|\, t, z\,|\!\rangle\,|\!\rangle)\ (f\ z) \\
\mathbf{P}^{n+1}N &= \mathbf{P}_{f_{n+1},s_{n+1},t_{n+1}}(\mathbf{P}^n N) \\
\mathbf{Q}_n &= \lambda f_{[n]}.\lambda s_{[n]}.\lambda t_{[n]}.\lambda x.\lambda y.\mathbf{P}^n\,\mathit{eqty}\,(x, y)
\end{aligned}$$

It is tedious but not difficult to see that all the relevant effects on coercions which is exploited in our construction above are also present under this encoding, using the encoded pair-type $(\tau_1 \to \tau_2 \to \sigma) \to \sigma$ (for arbitrary type σ) to encode the type $\tau_1 \times \tau_2$. In particular, G- and S-simplification still leaves an exponential number of observable 2-*crowns*, and Theorem 4.2.5 can therefore be used as before.

[1] This can be explained in terms of the Curry-Howard isomorphism, because one cannot define logical conjunction as a derived notion from implication in minimal logic (see, *e.g.*, [63].)

A.4 Proofs for Chapter 10

Proof of Theorem 10.2.3

Theorem 10.2.3 *For any constraint set C one has:*

1. *C is satisfiable in $\mathcal{T}_\Sigma$ if and only if C is satisfied by a C-shaped valuation*
2. *If C is satisfiable and α, β are non-trivially related in C, then $C \models \alpha \leq \beta$ if and only if it holds for any C-*shaped *valuation v that $v \models \alpha \leq \beta$ whenever $v \models C$.*

PROOF As for the first claim, if C is satisfiable in $\mathcal{T}_\Sigma$, then C is structural, and $C^\flat$ is satisfiable, by Lemma 10.1.5; by Lemma 8.3.3, in turn, this entails that $C^\flat$ is satisfiable in L. This shows the implication from left to right, and the other implication is obvious.

To prove the second claim, assume that it holds for any *C-shaped* valuation v' that $v' \models \alpha \leq \beta$ whenever $v' \models C$. We have $C \models \alpha \leq \beta$ if and only if $C^\flat \models [\alpha \leq \beta]_C$, by Lemma 10.1.5 (applicable since α, β are matching in C.) It is therefore sufficient to prove

$$C^\flat \models [\alpha \leq \beta]_C$$

In order to prove this, let $A \leq A' \in [\alpha \leq \beta]_C$. We must prove

$$C^\flat \models A \leq A' \tag{A.16}$$

where $A = \Theta_C(\alpha)(w)$, $A' = \Theta_C(\beta)(w)$ for some leaf w in $\Theta_C(\alpha)$ and $\Theta_C(\beta)$. By our assumption that α and β are non-trivially related in C, one of the follwing conditions is satisfied:

(i) A and A' are equivalent to a constant in $C^\flat$, or

(ii) neither A nor A' is equivalent to a constant in $C^\flat$.

Assume that (i) is the case. Let $v \models C^\flat$. Then Lemma 8.3.3 entails that $\widehat{v} \models C^\flat$ and hence (by Lemma 10.1.5) we have $\widehat{v} \circ \Theta_C \models C$. The valuation $\widehat{v} \circ \Theta_C$ is C-shaped, because $\widehat{v}$ is a valuation in L. Then, by our assumption that the entailment holds for C-shaped valuations, we get

$$\widehat{v} \circ \Theta_C \models \alpha \leq \beta \tag{A.17}$$

By (i) and $\widehat{v} \models C^\flat$, $\widehat{v}$ must map A and A' to constants in L, from which it follows that $\widehat{v}(A) = v(A)$ and $\widehat{v}(A') = v(A')$. It then follows from $(A.17)$ together with Lemma 10.1.5 (applied to $v \models C^\flat$) that $v \models A \leq A'$, thereby proving the entailment $(A.16)$.

Assume now that (ii) is the case. By (ii) and Lemma 8.3.3, the entailment $(A.16)$ is equivalent to $C^\flat \models_L A \leq A'$. So let v be a valuation in L such that $v \models C^\flat$. By Lemma 10.1.5 we have $v \circ \Theta_C \models C$. The valuation $v \circ \Theta_C$ is C-shaped, because v is a valuation in L, and consequently we get

$$\widehat{v} \circ \Theta_C \models \alpha \leq \beta \tag{A.18}$$

by our assumption that the entailment holds for C-shaped valuations. As in the previous case, it follows from $(A.18)$ together with Lemma 10.1.5 that $v \models A \leq A'$, thereby proving $(A.16)$ in this case also. □

Proof of Lemma 10.2.4

Lemma 10.2.4 *Let C be satisfiable. If α and β are trivially related in C, then $C \not\models \alpha \leq \beta$.*

PROOF If the first non–triviality condition is violated, because, say, α does not occur in C, any satisfying valuation can map α to anything, and the entailment obviously does not hold in this case; suppose the second condition is violated, then some leaf of, say, $\Theta_C(\alpha)$ is forced to be mapped to a constant in L, but this is not the case for the corresponding leaf of $\Theta_C(\beta)$, and a satisfying valuation (of $\Theta_C(C)$) can map that leaf of $\Theta_C(\beta)$ to a non–constant (a tree), witnessing non–entailment; if the third condition is violated, then $\Theta_C(\alpha)$ and $\Theta_C(\beta)$ do not have the same shape, and by Theorem 10.2.3 we can satisfy C by a C-shaped valuation v, where $v(\alpha)$ and $v(\beta)$ do not match, so $v(\alpha) \leq v(\beta)$ cannot hold, by Lemma 2.1.1, again resulting in non–entailment. □

Proof of Lemma 10.3.1

Lemma 10.3.1 *Let C be weakly unifiable. If V is a non-empty subset of $\mathsf{Var}(C^\top)$ and V is matching with respect to $C^\top$, then*

$$\bigcap_{\alpha \in V} \Uparrow_{C^\top}(\alpha) \neq \emptyset$$

PROOF Let $s_\top$ be the shape map associated with $C^\top$. Notice that, for $\alpha \in \mathsf{Var}(C)$, one has

$$s_C(\alpha) = s_\Gamma(\alpha) = s_\Delta(\alpha) = s_\top(\alpha)$$

and for $\gamma \in \Gamma$ one has

$$s_\Gamma(\gamma) = s_\Delta(\gamma) = s_\top(\gamma)$$

The effect of adding the new equations in C_Δ to C_Γ is just to identify all variables $\gamma \in \Gamma$ that have the same shape and therefore *define* the same shape. Using these observations, it is easy to see that every valuation satisfying C_Γ extends uniquely to a valuation satisfying $C^\top$. The set C_Γ, in turn, is obviously equivalent to C' given by

$$C' = C \cup \{\alpha \leq \mathrm{top}^s \mid \alpha \in \mathsf{Var}(C), s = s_C(\alpha)\}$$

in the sense that any valuation satisfying C' extends uniquely to a valuation satisfying C_Γ. Hence, in total, $C^\top$ is just another version of C' where all inequalities are simple and all variables γ involved in defining the elements top^s have been identified, whenever they have the same shape.

Since C is weakly unifiable, its shape map s_C is defined. We have

$$\mathsf{Var}(C^\top) = \mathsf{Var}(C) \cup \Gamma \cup \Delta$$

We first show:

$$\forall \alpha \in \mathsf{Var}(C) \cup \Gamma.\ \alpha \leq \delta_{s_\top(\alpha)} \in C^\top \tag{A.19}$$

To prove $(A.19)$, suppose first that $\alpha \in \mathsf{Var}(C)$. Let $s = s_C(\alpha)$. Then $\alpha \leq \gamma_s \in C_\Gamma$. Since $s = s_\Gamma(\alpha)$, we have $\gamma = \delta_s \in C_\Delta$. It follows that $\alpha \leq \delta_s \in C^\top$, since $C^\top$ is closed. Because $s = s_\top(\alpha)$, the claim follows. Suppose now that $\gamma \in \Gamma$, and let $s = s_\Gamma(\gamma)$. Then $\gamma = \delta_s \in C_\Gamma$. Since $s = s_\top(\gamma)$, the result follows.

Finally, suppose that $\delta_s \in \Delta$. Then $\gamma = \delta_s \in C_\Gamma$ for some γ. Since γ is part of a contractive system of equations, there is some non-variable type τ such that $\gamma \leq \tau \in C_\Gamma$. Hence, $\delta_s \leq \tau \in C^\top$, because $C^\top$ is closed.

Now let s be the common shape (under $s_\top$) with respect to $C^\top$ of all the variables in V. Then, for any $\alpha \in V \setminus \Delta$ we have $\alpha \leq \delta_s \in C^\top$, by $(A.19)$. If $\alpha \in V \cap \Delta$, then $\alpha = \delta_s$, because δ_s is the only variable in Δ with shape s in $C^\top$. Since $\delta_s \leq \tau \in C^\top$ for some non-variable type τ, it follows that,

there is a non-variable type τ such that for all $\alpha \in V$ we have $\alpha \leq \tau \in C^\top$. Then

$$\tau \in \bigcap_{\alpha \in V} \Uparrow_{C^\top} (\alpha)$$

thereby proving the lemma. □

Proof of Corollary 10.3.2

Corollary 10.3.2 *For any weakly unifiable constraint set C one has:*

1. *C is satisfiable if and only if $C^\top$ is satisfiable*
2. *If C is satisfiable and α, β are non-trivially related in C, then $C \models \alpha \leq \beta$ if and only if $C^\top \models \alpha \leq \beta$.*

PROOF As for the first claim, only the implication from left to right is not trivial. So assume C satisfiable. Then C is satisfied by a C-shaped valuation, by Theorem 10.2.3. It is easy to see that any C-shaped valuation v satisfying C must also satisfy $v(\alpha) \leq \mathrm{top}^{s(\alpha)}$ for all α in C (observe that $v(\alpha) \in L_{s(\alpha)}$, and recall that $\mathrm{top}^{s(\alpha)}$ is the top element in this lattice.) The second claim follows from the same observation together with Theorem 10.2.3. □

Proof of Lemma 10.3.3

In order to prove Lemma 10.3.3, we first prove two technical lemmas. The following lemma says that, starting $\mathcal{A}^C_D$ in a set matching wrt. C, one reaches only sets which are again matching wrt. C:

Lemma A.4.1 *Let w be a string in $\mathbf{A}^*$ accepted by $\mathcal{A}^C_D(q_0)$ with C weakly unifiable and q_0 matching with respect to C, and suppose that $\widehat{\delta}_D(q_0, w) = q$. Then q is again matching with respect to C.*

PROOF By induction on $|w| \geq 0$.

For the base case, the statement is just that q_0 is matching wrt. C, which we have assumed.

Suppose for the inductive step that $w = fw'$. Let $q' = \delta_D(q_0, f)$, so that $q = \widehat{\delta}_D(q', w')$. Then (since a transition on f is defined) it must be that $q_0 = \{\alpha_i \times \alpha'_i\}_{i \in I}$, and since q_0 is matching wrt. C, we must have $\{\alpha_i\}_{i \in I}$ matching wrt. C also. Let s be the common shape of all the α_i wrt. C.

For each i and $\tau \in \Uparrow_C (\alpha_i)$ we evidently have $s_C(\alpha_i) = s_C(\tau)$; it follows that $s_C(\tau) = s$ for all $\tau \in \Uparrow_C (\alpha_i)$ and all i, and therefore the set $\bigcup_i \Uparrow_C (\alpha_i)$ is matching wrt. C. But this set is just q', and induction hypothesis applied to q' now proves the claim. Other cases are similar to this one and are left out. □

Lemma A.4.2 *Let w be a string in $\mathbf{A}^*$ accepted by $\mathcal{A}_D^{C^\top}(q_0)$ with C weakly unifiable and q_0 nonempty and matching with respect to $C^\top$, and suppose that $\widehat{\delta}_D(q_0, w) = q$. Then q is again nonempty and matching with respect to $C^\top$.*

PROOF That q is matching wrt. $C^\top$ follows from Lemma A.4.1.

To prove non-emptyness, we proceed by induction on $|w| \geq 0$, and the base case is trivial, since the statement is true of q_0 by assumption.

Assume for the inductive case that $w = dw'$ (other cases are similar and are left out), and let $q' = \delta_D(q_0, d)$, so that $q = \widehat{\delta}_D(q', w')$. Then (since a transition on d is defined) it must be that $q_0 = \{\alpha_i \to \alpha_i'\}_{i \in I}$, where the set $\{\alpha_i\}_{i \in I}$ is non-empty and matching wrt. $C^\top$ because q_0 is. We have $q' = \bigcap_i \Uparrow_{C^\top} (\alpha_i)$, where q' is matching wrt. $C^\top$ by Lemma A.4.1; since the set $\{\alpha_i\}_{i \in I}$ is non-empty and matching, Lemma 10.3.1 shows that q' is non-empty. We have now shown that q' is non-empty and matching wrt. $C^\top$, and induction hypothesis applied to q' proves the lemma. □

We can now prove:

Lemma 10.3.3 *If C is weakly unifiable and $\alpha \in \mathsf{Var}(C)$, then $t_\wedge^C(\alpha)$ is a well defined term automaton (i.e., its labeling function is defined on all states reachable from the start state of the automaton.)*

PROOF Because C is weakly unifiable, the singleton sets $\{\alpha\}$ ($\alpha \in \mathsf{Var}(C)$) are necessarily non-empty and matching with respect to C, and therefore Lemma 10.3.1 shows that the start states q_α are non-empty; they are also matching with respect to C, because C is weakly unifiable. It then follows from Lemma A.4.2 that $\ell_\wedge$ is well-defined on all states reachable from q_α in $\mathcal{A}_D^{C^\top}$. □

Proof of Lemma 10.4.1

In order to prove Lemma 10.4.1, we first prove two small lemmas. Let $T(C^\top)$ denote the set of non–variable subterms occurring in $C^\top$.

Lemma A.4.3 *Let C be weakly unifiable, and let q_1 and q_2 be non-empty subsets of $T(C^\top)$ each of which is matching with respect to $C^\top$. If w is accepted by $\mathcal{A}_D^{C^\top}(q_1)$ and $\mathcal{A}_D^{C^\top}(q_2)$, then*

$$q_1 \subseteq q_2 \text{ implies } \widehat{\delta}_D(q_1, w) \subseteq^w \widehat{\delta}_D(q_2, w)$$

PROOF By induction on the length of w, as in [60]. □

Our labeling function is anti-monotone:

Lemma A.4.4 *Let q_1 and q_2 be sets of terms for which $\ell_\wedge$ is defined, $b \in \{0, 1\}$ a polarity.*

Then

$$q_1 \subseteq^b q_2 \text{ implies } \ell_\wedge(q_2) \leq^b \ell_\wedge(q_1)$$

PROOF The claim follows from the fact that the greatest lower bound of a smaller set results in a larger element. □

We can now prove

Lemma 10.4.1 *Suppose that C is consistent (weakly unifiable and ground consistent). Let $\Uparrow$ denote the function $\Uparrow_{C^\top}$. Let $\tau_1 \leq \tau_2 \in C^\top$ and $q_1 = \Uparrow(\tau_1)$, $q_2 = \Uparrow(\tau_2)$. Then , for any string $w \in \mathbf{A}^*$ accepted by both $\mathcal{A}_D^{C^\top}(q_1)$ and $\mathcal{A}_D^{C^\top}(q_2)$ one has*

$$\ell_\wedge(\widehat{\delta}_D(q_1, w)) \leq^w \ell_\wedge(\widehat{\delta}_D(q_2, w))$$

PROOF First notice that $C^\top$ is weakly unifiable, ground consistent and closed: The set $C^\top$ arises from C by adding to C (representations of) inequalities of the form $\alpha \leq \mathrm{top}^s$, $s = s_C(\alpha)$. This results in a ground consistent set if C was already ground consistent, because the only new atomic inequalities thereby introduced in $C^\top$ must be of the form $A \leq \top$ or $\bot \leq A$.

Hence, since C is weakly unifiable and ground-consistent, so is $C^\top$. Moreover, $C^\top$ is by definition closed.

Since we assume that C is given in simple form, we know that one of τ_1 and τ_2 is a variable, and that the depth of τ_1 and τ_2 is at most 1. Notice also that, since $C^\top$ is weakly unifiable, one has $\Uparrow(\alpha)$ matching with respect to $C^\top$ for all $\alpha \in \mathsf{Var}(C)$, and therefore (by Lemma 10.3.1) $\Uparrow(\alpha) \neq \emptyset$ and again matching with respect to $C^\top$. It follows that $\ell_\wedge$ is well defined on such sets. This observation is tacitly used throughout the remainder of this proof.

We proceed by cases over the form of w and $\tau_1 \leq \tau_2 \in C^\top$.

Case (1). Assume first that τ_1 is a variable, $\tau_1 = \alpha$. In this case, if $\tau_2 = \beta$ is also a variable, we have $\Uparrow(\beta) \subseteq \Uparrow(\alpha)$. Therefore, Lemma A.4.3 shows that

$$\widehat{\delta}_D(\Uparrow(\beta), w) \subseteq^w \widehat{\delta}_D(\Uparrow(\alpha), w)$$

and together with Lemma A.4.4 this entails

$$\widehat{\delta}_D(\Uparrow(\alpha), w) \leq^w \widehat{\delta}_D(\Uparrow(\beta), w)$$

thereby proving the lemma in this case. If $\tau_2 = b$ is a constant in L, we must have $w = \epsilon$ and $\Uparrow(\alpha) \subseteq L$ (and, by definition, $\Uparrow(b) = \{b\}$). Here $b \in \Uparrow(\alpha)$, and it follows that

$$\ell_\wedge(\Uparrow(\alpha)) \leq_L \ell_\wedge(\Uparrow(b))$$

thereby proving the lemma in this case. Finally, if τ_2 is a constructed type, assume $\tau_2 = \beta_1 \to \beta_2$ (other cases for τ_2 are handled similarly and are left to the reader.) We have

$$\Uparrow(\beta_1 \to \beta_2) = \{\beta_1 \to \beta_2\} \subseteq \Uparrow(\alpha)$$

and Lemma A.4.3 entails

$$\widehat{\delta}_D(\Uparrow(\beta_1 \to \beta_2), w) \subseteq^w \widehat{\delta}_D(\Uparrow(\alpha), w)$$

and the result now follows from Lemma A.4.4.

Case (2). Now assume that τ_1 is not a variable and $\tau_2 = \beta$ is a variable. τ_1 is either a constant or a constructed type.

Case (2.1). Assume first that $\tau_1 = b$ is a constant in L. The inequality under consideration is therefore $b \leq \beta$. Accordingly, $\Uparrow(\beta) \subseteq L$ (by weak unifiability), and $w = \epsilon$ must be the case. Since $C^\top$ is ground consistent, we have

$$\forall b' \in \Uparrow(\beta).\ b \leq_L b'$$

It follows that

$$\ell_\wedge(\Uparrow(b)) = b \leq_L \ell_\wedge(\Uparrow(\beta))$$

since $\ell_\wedge$ acts as the greates lower bound operation in L. This proves the lemma in this case.

Case (2.2). If τ_1 is not a constant, then it is a constructed type. We assume that τ_1 is a function type, $\tau_1 = \alpha_1 \to \alpha_2$ (the other possibility for τ_1 is similar and is left to the reader.) We proceed by cases over the form of w.

Assume first that $w = \epsilon$. The set $\Uparrow(\beta)$ must be a non-empty set of function types (by weak unifiability of $C^\top$), so we can write $\Uparrow(\beta) = \{\beta_i \to \beta_i'\}_i$. On the other hand, $\Uparrow(\alpha_1 \to \alpha_2) = \{\alpha_1 \to \alpha_2\}$, and therefore

$$\ell_\wedge(\Uparrow(\alpha_1 \to \alpha_2)) \ = \to = \ \ell_\wedge(\Uparrow(\beta))$$

proving the lemma in this case.

Now assume $w \neq \epsilon$. We show how to reason in the case where $w = dw'$, in order to demonstrate how contra-variance is handled. Remaining cases are left to the reader (the principles of the argument can be seen from what follows.) We consider the inequality $\alpha_1 \to \alpha_2 \leq \beta$. We have $\Uparrow(\alpha_1 \to \alpha_2) = \{\alpha_1 \to \alpha_2\}$, and $\Uparrow(\beta)$ is (by weak unifiability) a non-empty set of function types, so we can write $\Uparrow(\beta) = \{\beta_i \to \beta_i'\}_i$. We must show that

$$\ell_\wedge(\widehat{\delta}_D(\Uparrow(\alpha_1 \to \alpha_2), dw')) \leq^{dw'} \ell_\wedge(\widehat{\delta}_D(\Uparrow(\beta), dw')) \tag{A.20}$$

Let

$$q_1 = \delta_D(\Uparrow(\alpha_1 \to \alpha_2), d)$$

and

$$q_2 = \delta_D(\Uparrow(\beta), d)$$

Then, by the definition of $\widehat{\delta}_D$ together with change of polarity in passing from w' to dw', we have $(A.20)$ equivalent to

$$\ell_\wedge(\widehat{\delta}_D(q_2, w')) \leq^{w'} \ell_\wedge(\widehat{\delta}_D(q_1, w')) \tag{A.21}$$

which we now proceed to prove.

Since $C^\top$ is transitively closed, we have $\alpha_1 \to \alpha_2 \leq \beta_i \to \beta_i' \in C^\top$ for all i. Because $C^\top$ is closed under decomposition, this implies in turn that $\beta_i \leq \alpha_1 \in C^\top$ for all i. It follows that $\Uparrow(\alpha_1) \subseteq \Uparrow(\beta_i)$ for all i, and hence

$$\Uparrow(\alpha_1) \subseteq \bigcap_i \Uparrow(\beta_i)$$

By Lemma 10.3.1 and the fact that $C^\top$ is weakly unifiable, both sets mentioned above are non-empty and matching with respect to $C^\top$. By the definition of the automaton $\mathcal{A}_D^{C^\top}$, we have

$$q_1 = \Uparrow(\alpha_1)$$

and

$$q_2 = \bigcap_i \Uparrow(\beta_i)$$

so we have just shown that $q_1 \subseteq q_2$, with q_1 and q_2 both non-empty and matching wrt. $C^\top$. Then Lemma A.4.3 shows that

$$\widehat{\delta}_D(q_1, w') \subseteq^{w'} \widehat{\delta}_D(q_2, w')$$

and therefore, by Lemma A.4.4 we get

$$\ell_\wedge(\widehat{\delta}_D(q_2, w')) \leq^{w'} \ell_\wedge(\widehat{\delta}_D(q_1, w'))$$

which is just the property $(A.21)$ we aimed to prove. □

Bibliography

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] A. Aiken, F. Henglein, and J. Rehof. Subtyping and the cubic time bottleneck. Preliminary draft, August 1997.

[3] A. Aiken and E.L. Wimmers. Type inclusion constraints and type inference. In *Proceedings FPCA '93, Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 31–42, June 1993.

[4] A. Aiken, E.L. Wimmers, and J. Palsberg. Optimal representations of polymorphic types with subtyping. In *Proceedings TACS '97, Theoretical Aspects of Computer Software, Sendai, Japan*, pages 47–77. Springer Lecture Notes in Computer Science, vol. 1281, September 1997.

[5] R. Amadio and L. Cardelli. Subtyping recursive types. In *Proc. 18th Annual ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 104–118. ACM Press, January 1991.

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

[7] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[8] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 117–309. Oxford University Press, 1992.

[9] M. Benke. Efficient type reconstruction in the presence of inheritance. In *Mathematical Foundations of Computer Science (MFCS)*, pages 272–280. Springer Verlag, LNCS 711, 1993.

[10] M. Benke. Some complexity bounds for subtype inequalities. Technical Report TR 95-20 (220), Warsaw University, Institute of Informatics, Warsaw University, Poland, December 1995.

[11] P. A. Bloniarz, H. B. Hunt, and D. J . Rosenkrantz. Algebraic structures with hard equivalence and minimization problems. Technical Report 82-3, SUNY, March 1982.

[12] P. A. Bloniarz, H. B. Hunt, and D. J. Rosenkrantz. Algebraic structures with hard equivalence and minimization problems. *Journal of the ACM*, 31(4):879–904, October 1984.

[13] F. Bourdoncle and S. Merz. Type checking higher–order polymorphic multi–methods. In *Proceedings POPL '97, 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France*, pages 301–315. ACM, January 1997.

[14] W. Charatonik and A. Podelski. Set constraints with intersection. In *Proceedings LICS '97, Twelfth Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland*, pages 362–372. IEEE Computer Society Press, June 1997.

[15] S. S. Cosmadakis. The word and generator problem for lattices. *Information and Computation*, 77:192–217, 1988.

[16] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[17] P. Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, Xerox Parc, February 1990.

[18] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1984. Technical Report CST-33-85 (1985).

[19] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, January 1982.

[20] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[21] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings OOPSLA '95*, 1995.

[22] M. Fahndrich and A. Aiken. Making set-constraint program analyses scale. In *Workshop on Set Constraints, Cambridge MA*, 1996.

[23] M. Fahndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings PLDI '98, ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Canada*, June 1998. To appear.

[24] C. Flanagan. Personal communication, March 1997.

[25] C. Flanagan and M. Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. ACM, June 1997.

[26] A. Frey. Satisfiability of subtype inequalities in structural infinite terms over lattices in PTIME. Unpublished note, April 1997.

[27] A. Frey. Satisfying subtype inequalities in polynomial space. In *Proceedings SAS '97, International Static Analysis Symposium, Paris, France 1997*, pages 265–277. Springer Lecture Notes in Computer Science, vol. 1302, 1997.

[28] Y. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proc. Int'l J't Conf. on Theory and Practice of Software Development*, pages 167–183, Barcelona, Spain, March 1989. Springer-Verlag.

[29] Y. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science (TCS)*, 73:155–175, 1990.

[30] M. Garey and D. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[31] N. Heintze and D. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings LICS '97, Twelfth Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland*, pages 342–351. IEEE Computer Society, June 1997.

[32] F. Henglein. Preunification and precongruence closure: Algorithms and applications. Preliminary draft, September 1992.

[33] F. Henglein and J. Rehof. The complexity of subtype entailment for simple types. In *Proceedings LICS '97, Twelfth Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland*, pages 352–361. IEEE Computer Society Press, June 1997.

[34] F. Henglein and J. Rehof. Constraint automata and the complexity of recursive subtype entailment. In *Proceedings ICALP '98, 5th International Colloquium on Automata, Languages, and Programming. Aalborg, Denmark.* (To appear), July 1998.

[35] J.R. Hindley. *Basic Simple Type Theory.* Cambridge Tracts in Theoretical Computer Science, 42, Cambridge University Press, 1997.

[36] R. Hindley and J. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[37] M. Hoang and J.C. Mitchell. Lower bounds on type inference with subtypes. In *Proc. 22nd Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 176–185. ACM Press, 1995.

[38] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[39] H. B. Hunt, D. J. Rosenkrantz, and P.A. Bloniarz. On the computational complexity of algebra on lattices. *SIAM Journal of Computing*, 16(1):129–148, February 1984.

[40] M. P. Jones. *Qualified Types: Theory and Practice.* Cambridge University Press, 1994.

[41] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proc. ACM Conf. on LISP and Functional Programming (LFP), San Francisco, California*, pages 193–204. ACM Press, June 1992. also in LISP Pointers, Vol. V, Number 1, January-March 1992.

[42] P. Kanellakis, H. Mairson, and J.C. Mitchell. Unification and ML type reconstruction. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic — Essays in Honor of Alan Robinson.* MIT Press, 1991.

[43] G. Kildall. A unified approach to global program optimization. *Proc. At tCM Symp. on Principles of Programming Languages (POPL)*, 1973.

[44] D. Kozen, J. Palsberg, and M.I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994.

[45] Dexter Kozen, Jens Palsberg, and Michael Schwartzbach. Efficient recursive subtyping. In *Proc. 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 419–428. ACM, ACM Press, January 1993.

[46] D. J. Lillie. Conjunctive subtyping. In *Proceedings FPCA '93, ACM SIGPLAN/SIGARCH Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 42–52, June 1993.

[47] P. Lincoln and J.C. Mitchell. Algorithmic aspects of type inference with subtypes. In *Proc. 19th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programmin Languages (POPL), Albuquerque, New Mexico*, pages 293–304. ACM Press, January 1992.

[48] S. Marlow and P. Wadler. A practical subtyping system for erlang. In *2nd International Conference on Functional Programming, Amsterdam.* ACM, June 1997.

[49] E. Melski and T. Reps. Interconvertability of set constraints and context–free language reachability. In *Proceedings PEPM '97, ACM SIGPLAN Symposium on Partial Evaluation and Semantics–Based Program Manipulation.* ACM Press, 1997.

[50] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[51] J.C. Mitchell. Coercion and type inference (summary). In *Proc. 11th ACM Symp. on Principles of Programming Languages (POPL)*, pages 175–185, 1984.

[52] J.C. Mitchell. Polymorphic type inference and containment. *Information and Control*, 76:211–249, 1988.

[53] J.C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, July 1991.

[54] J.C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[55] C. Mossin. *Flow Analysis of Typed Higher–Order Programs.* PhD thesis, DIKU Technical Report No. 97/1, Dept. of Computer Science, University of Copenhagen, 1997.

[56] P. O'Keefe and M. Wand. Type inference for partial types is decidable. In B. Krieg-Brückner, editor, *Proc. 4th European Symp. on Programming (ESOP), Rennes, France*, pages 408–417. Springer, February 1992. Lecture Notes in Computer Science, Vol. 582.

[57] 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.

[58] J. Palsberg, M. Wand, and P. O'Keefe. Type inference with nonstructural subtyping. BRICS Technical Report RC-95-33, Department of Computer Science, University of Aarhus, April 1995.

[59] F. Pottier. Simplifying subtyping constraints. In *Proceedings ICFP '96, International Conference on Functional Programming*, pages 122–133. ACM Press, May 1996.

[60] F. Pottier. Simplifying subtyping constraints (full version). Technical report, INRIA, Ecole Normale Superieure, October 1996.

[61] V. Pratt. Personal communication, May 1997.

[62] V. Pratt and J. Tiuryn. Satisfiability of inequalities in a poset. *Fundamenta Informaticae*, 28:165–182, 1996.

[63] D. Prawitz. *Natural deduction.* Almquist & Wiksell, Uppsala 1965.

[64] J. Rehof. Minimal typings in atomic subtyping. Technical Report D-278, DIKU, Dept. of Computer Science, University of Copenhagen, Denmark. Available at http://www.diku.dk/research-groups/topps/personal/rehof/publications.html, 1996.

[65] J. Rehof. Minimal typings in atomic subtyping. In *Proceedings POPL '97, 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France*, pages 278–291. ACM, January 1997.

[66] J. Rehof and T. Mogensen. Tractable constraints in finite semilattices. In *Proceedings SAS '96, International Static Analysis Symposium, Aachen, Germany, September 1996*. Springer Lecture Notes in Computer Science, vol. 1145, 1996. Journal version to appear in *Science of Computer Programming*, 1998.

[67] G. S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[68] S. F. Smith. Constrained types: A personal perspective. `http://www.cs.jhu.edu/hog/constrained.ps`, September 1996.

[69] S. Thatte. Type inference with partial types. In *Proc. Int'l Coll. on Automata, Languages and Programming (ICALP)*, pages 615–629, 1988. Lecture Notes in Computer Science, vol 317.

[70] S. R. Thatte. Type inference with partial types. *Theoretical Computer Science*, 124(1):127–148, February 1994.

[71] J. Tiuryn. Subtype inequalities. In *Proc. 7th Annual IEEE Symp. on Logic in Computer Science (LICS), Santa Cruz, California*, pages 308–315. IEEE Computer Society Press, June 1992.

[72] J. Tiuryn and M. Wand. Type reconstruction with recursive types and atomic subtyping. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. Theory and Practice of Software Development (TAPSOFT), Orsay, France*, volume 668 of *Lecture Notes in Computer Science*, pages 686–701. Springer-Verlag, April 1993.

[73] V. Trifonov and S. Smith. Subtyping constrained types. In *Proceedings SAS '96, Static Analysis Symposium, Aachen, Germany*, pages 349–365. Springer, 1996. Lecture Notes in Computer Science, vol.1145.

[74] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, X:115–122, 1987.