Interactive and Automated Reasoning with Orthologic

Simon Guilloud

Abstract

L'usage des méthodes formelles en mathématiques et en génie logiciel croît, mais l'adoption à grande échelle des outils de vérification reste freinée par des barrières et des inefficacités. En particulier, la coNP-complétude du problème de validité pour la logique propositionnelle classique oblige les outils à utiliser des heuristiques soit incomplètes, soit avec une mauvaise complexité asymptotique. Les solveurs SAT et SMT ont démontré une grande efficacité en pratique, mais leur intégration dans les assistants de preuve est difficile, et leur opacité peut rendre leur comportement moins prévisible pour les utilisateurs finaux. Par ailleurs, les procédures de simplification et de normalisation sont omniprésentes, mais les plus courantes sont soit faibles (forme normale négative), ne préservent pas l'équivalence (transformation de Tseitin), ou sont coûteuses en temps de calcul (forme normale conjonctive ou disjonctive complète).

Dans cette thèse, nous proposons d'utiliser des algèbres légèrement plus faibles que les algèbres de Bool mais qui possèdent de bonnes propriétés computationnelles en tant qu'approximation du raisonnement propositionnel classique. Concrètement, nous introduisons le *raisonnement basé sur l'orthologique* (orthologic-based reasoning). L'orthologique est une logique non classique basée sur la classe algébrique des orthotreillis. Elle utilise les connecteurs $\neg$, $\wedge$ et $\vee$ des algèbres de Bool et partage la plupart de leurs axiomes, mais, crucialement, ne postule pas la loi de distributivité. Cela signifie que tous les théorèmes de l'orthologique sont des théorèmes de la logique classique, mais pas l'inverse. Cet affaiblissement nous permet de développer des algorithmes en temps polynomial, tout en conservant une grande partie de l'expressivité de la logique classique et en offrant des garanties de complétude bien caractérisées.

Nos contributions couvrent l'ensemble du spectre de la vérification : fondations théoriques, algorithmes, applications et implémentations. Nous proposons deux nouveaux algorithmes clés : (1) un algorithme de normalisation en $\mathcal{O}(n^2)$ qui transforme chaque formule en la plus petite formule dans sa classe d'équivalence en orthologique, et (2) un algorithme en $\mathcal{O}(n^2(1+|A|))$ qui décide la validité d'une formule sous les axiomes de l'orthologique et un ensemble A d'hypothèses non logiques. Pour analyser ces algorithmes, nous développons la théorie de la preuve de l'orthologique et prouvons des propriétés telles que la propriété de sous-formule et la propriété d'interpolation. Nous formalisons les résultats clés dans l'assistant de preuve Rocq et démontrons plusieurs applications. Entre autres, nous présentons un système de types avec des types d'union `|` et d'intersection `&` où la relation de sous-typage est décidée en utilisant notre algorithme d'orthologique. Nous implémentons également la normalisation orthologique dans le vérificateur de programmes Stainless, réduisant la taille des conditions de vérification et améliorant l'efficacité du cache.

Enfin, nous présentons Lisa, un nouvel assistant de preuves dont le noyau logique utilise la normalisation orthologique pour considérer automatiquement les formules équivalentes comme interchangeables. Lisa repose sur des fondations ensemblistes traditionnelles visant à convaincre la communauté mathématique, et offre un langage de preuves compatible avec le langage de programmation Scala, facilitant l'implémentation de l'automatisation des preuves. Lisa accorde une importance particulière à l'*interopérabilité* : nous encodons la logique d'ordre supérieure dans la théorie des ensembles et démontrons que cela permet de simuler des preuves du système HOL Light. Nous présentons également SC-TPTP, un format et un utilitaire pour l'échange de preuves entre systèmes de preuve basés sur la logique du premier ordre. Ce format est intégré dans Lisa et cinq autres assistants de preuve automatisés ou interactifs.

Abstract

The use of formal verification for mathematics and software is growing, yet the mainstream adoption of verification systems remains hindered by barriers and inefficiencies. The coNP-completeness of the validity problem for classical propositional logic in particular forces tools to rely on heuristics that either are incomplete or have impractical worst-case complexity. SAT and SMT solvers have proved highly effective in practice, but their integration into proof assistants is difficult, and their opacity can make their behaviour less predictable to end users. At the same time, simplification and normalization procedures are ubiquitous, but the most common ones either are weak (negation normal form), do not preserve equivalence (Tseitin's transform) or are computationally expensive (full conjunctive or disjunctive normal form).

In this thesis, we propose to use algebras that are slightly weaker than Boolean algebras but with good computational properties as an approximation of classical propositional reasoning. Concretely, we introduce *orthologic-based reasoning*. Orthologic is a non-classical logic based on the algebraic class of ortholattices. It uses the connectives $\neg$, $\wedge$ and $\vee$ of Boolean algebras and shares most of its axioms, but crucially does not assume the distributivity law. This means that all orthologic theorems are theorems of classical logic, but not conversely. This small weakening allows us to design polynomial-time algorithms, while retaining a large part of the expressivity of classical logic and offering well-characterized completeness guarantees.

Our contributions span the entire spectrum of verification: theoretical foundations, algorithms, applications and implementations. We propose two key new algorithms: (1) an $\mathcal{O}(n^2)$ normalization algorithm that maps formulas to the smallest formula in its orthologic equivalence class, and (2) an $\mathcal{O}(n^2(1+|A|))$ algorithm for deciding the validity of a formula under orthologic axioms and an arbitrary set A of non-logical assumptions. To demonstrate the correctness of these algorithms, we develop the proof theory of orthologic and prove properties such as the subformula property and the interpolation property. We formalize key results in the Rocq proof assistant and demonstrate multiple applications. Among other things, we present a type system with union `|` and intersection `&` types where the subtyping relation is decided using our orthologic algorithm. We also implement orthologic normalization in the Stainless verifier, reducing the size of verification conditions and improving the cache hit rate.

Finally, we present Lisa, a new proof assistant whose logical core uses orthologic normalization to automatically consider equivalent formulas interchangeable. Lisa relies on mainstream set-theoretic foundations aimed at appealing to the mathematics community, and offers a proof script language that is fully compatible with Scala, enabling easy implementation of proof automation. Lisa has a core focus on *interoperability*: we show an embedding of higher-order logic that enables simulation of proofs from the HOL Light proof system, and we present SC-TPTP, a format and toolchain for exchanging proofs between first-order logic proof systems; it is integrated into Lisa and five other automated or interactive theorem provers.

Thanks

Completing a PhD is at times a challenging journey and would not have been possible without support.

I express my sincere gratitude to my advisor Viktor for five and a half years of teaching, guidance and interesting discussions. I believe you taught me how to do science as well as it was possible. Mission accomplished!

I am also thankful to everyone with whom I have had the chance to collaborate, and in particular my co-authors. Special thanks to Clément for his repeated great advice and guidance. Special thanks also to Sankalp for so many interesting and/or entertaining discussions; it truly was an honor and a pleasure to share a whiteboard with you.

Great thanks to my friends and family for their encouragement. Each of you is, in some way or another, responsible for some of the events that led to this thesis.

Finally, deepest thanks to my fiancée Natalia for her unfaltering support and patience.

Contents

1 Introduction

Formal verification is more relevant than ever. To further enable widespread adoption, we must strive to make verification systems more *efficient* and *predictable*.

The mathematical community is increasingly adopting interactive theorem provers and mechanized proof systems in general, as the difficulty and specialization of state-of-the-art mathematical research make it hard for the community to review and certify cutting-edge, paper-written proofs. Thanks to progress in theorem provers and huge team efforts, theorems by Fields medallists have been verified shortly after, or even before, their publication [89]. Yet these formalizations remain expensive: Automation can be powerful but opaque, solver calls can be brittle, and small "obvious" logical steps can still require substantial proof effort.

Similarly, our societies increasingly rely on software systems, making these systems key targets for nefarious actors. The security vulnerabilities enabling these attacks can be eliminated by formal verification, but we remain unable to prove software correct at large scale. While large language models promise to make formal verification more accessible to users, they cannot be independently trusted. Hence, rather than replacing formal reasoning, they only highlight the need for efficient and predictable building blocks for verification systems.

One especially important fragment of automated reasoning is propositional reasoning: it is everywhere in verification tasks. SAT-based hardware verification reduces circuit correctness to large Boolean formulas. Program verifiers turn programs and their specifications into large verification conditions, while control-flow constructs (such as **`if`**/**`else`**/**`while`**, pattern matching, short-circuit semantics and exceptions) generate additional Boolean constraints. Analysis of security policies boils down to asking whether some relation between configurations of flags, permissions and protocol states can or cannot be satisfied. In type systems with union and intersection types, deciding whether `A & B` is a subtype of `A & (A | C)` is essentially equivalent to deciding whether the propositional implication $A \wedge B \implies A \wedge (A \vee C)$ is valid. Even when the hard part of the problem lies in a different domain (for example, arithmetic, recursive functions or concurrency models), propositional logic acts as a glue that connects statements. Verification systems then have to manipulate, simplify and solve the resulting formulas. In proof assistants, for example, it is common that a theorem loosely but not exactly

matches a goal or the assumption of another theorem: maybe a theorem states $A \wedge B$ while another lemma states $(B \wedge A) \implies C$, or maybe one theorem states $\forall x.\ \neg P(x)$ while another assumes $\neg\exists x.\ P(x)$. The proof assistant would not accept combining them directly, but require additional steps to reorder the conjuncts or rewrite the quantifiers. These are intuitively obvious transformations, but they cost time and make proofs longer, hampering the adoption of formal proofs. It would be desirable to automatically handle such transformations.

Another important task that verification systems need to perform is simplifying and normalizing formulas. As an example, a simple way to check if a formula ϕ is satisfiable (that is, to build a simple SAT solver) is the DPLL algorithm: substitute a variable by $\top$ and $\bot$ in ϕ, then simplify the two resulting formulas, and recursively check the satisfiability of the simplified formulas. If either is satisfiable, then so is ϕ. Normalization is also useful for *caching*: a verifier may check if a formula ϕ has already been solved earlier, and only call an external solver if it has not. Normalization allows to increase the hit rate by detecting early that ϕ is equivalent to a previously solved formula, even if not syntactically equal. The most commonly used normal forms are negation normal form (NNF) and conjunctive/disjunctive normal forms (CNF/DNF). But the NNF is fairly weak, and CNF/DNF are exponentially costly to compute in the worst case[1]. This raises a natural question: is there a normal form for propositional formulas that would be stronger than the NNF, but still computable in polynomial time?

All of these transformations happen inside complex verification pipelines: they combine a front-end, encodings, dedicated tools and clever algorithms. This generates engineering constraints of predictability and composability. Concretely, tool builders and users need components that interact robustly, with precise completeness and time complexity guarantees, so that failures and slowdowns can be understood and fixed. A well-known example where these principles sometimes fail is the instability of SMT solvers. Small syntactic variations or the addition of redundant hypotheses can drastically change the performance, or even determine whether the solver succeeds at all [31, 49]. The impact of such issues is exacerbated when tools are chained together. For example, a program verifier might use an SMT solver or a solver for constrained Horn clauses as a backend to check that some bad state is unreachable, and the user writes intermediate steps towards the verification goal to "guide" the underlying solver. When it fails to prove a goal, the user adds steps of lower and lower granularity. However, it can happen that even when the proof has reached the lowest possible granularity (to give a simple example, let us say that the assumptions are A and $A \implies B$ and the goal is B), the solver's heuristic still does not find the correct way to combine the facts[2]. It is then effectively impossible to complete the proof. In practice, this creates a gap between what is "obviously" derivable and what the tool can reliably discharge.

More broadly, unpredictability is a hindrance throughout the engineering workflow. Debugging itself is fragile when robustness is lacking, since debugging actions (inspecting

[1] The Tseitin transform takes linear time but only preserves satisfiability, not equivalence.

[2] I have observed such behaviour myself, and it is also reported, for example, in [53]

intermediate goals, isolating parts of the proof) may change the behaviour of tools, and this is made even more difficult by the fact that solvers typically cannot report on why a proof attempt fails. As a result, the cause of a failure is often unclear for the user. Is it due to a bug in the toolchain or a limitation of the underlying solver? A mistake in the user's code, proof or logic? Or simply bad luck? Results and benchmarks are also difficult to reproduce consistently on different systems, hindering the fixing of bug reports and performance evaluation. Toolchain updates can introduce regressions where previously verified developments no longer go through. Together, these effects translate into significant time lost, frustration and ultimately a barrier to the adoption of formal methods at large scale.

While modern SAT/SMT solvers are highly optimized and extraordinarily effective in many practical applications, they are often at odds with predictability and efficiency, and hence using them as a default "black box" is best reserved for when their power is truly needed.

A key idea of this thesis is to replace complete but heavy propositional reasoning engines in parts of verification systems by *small, fast, trustworthy building blocks* with (1) polynomial-time worst-case guarantees, (2) well-characterized algebraic guarantees and (3) implementations that can themselves be formally verified.

Key outcomes. The results of this thesis span work *from algebraic proof theory, to efficient and verified algorithms, to practical verification systems design.* Concretely, our most significant contributions are as follows:

1. We develop the proof theory of ortholattices, an algebraic class that generalizes Boolean algebras.

2. We present an $\mathcal{O}(n^2)$ normalization algorithm for propositional formulas that is stronger than negation-normal form, and an $\mathcal{O}(n^2(1+|A|))$ algorithm for deciding entailment under a set A of assumptions.

3. We introduce *orthologic-based reasoning*, an approach to efficient and predictable propositional reasoning based on ortholattices, and we demonstrate its application in key verification systems: proof assistants, program verifiers and type systems.

4. We verify in the Rocq proof assistant key algorithms and theorems of orthologic-based reasoning, in particular the normalization and entailment algorithms.

5. We show that orthologic-based reasoning brings practical improvements: up to 40% reduction in solving time for benchmarks in the Stainless verifier, tactics orders of magnitude faster than counterparts in the standard library of Rocq.

6. We design and implement Lisa, a new proof assistant that integrates orthologic-based reasoning to address constraints of usability, efficiency, predictability and trust.

7. We publish SC-TPTP, a toolchain and format for interoperability of proof systems based on first-order logic, and we integrate it in Lisa and multiple proof assistants and automated solvers.

8. We implement a mechanized translation of HOL Light's higher-order logic into Lisa's set theory, fully verifying HOL Light's proofs from set-theoretic principles and enabling the use of HOL Light's libraries in Lisa.

1.1 Detailed overview

Orthologic-based reasoning

Orthologic is a sound approximation of classical propositional logic that corresponds algebraically to the class of ortholattices. Ortholattices share the signature of Boolean algebras (conjunction, disjunction and negation), but do not necessarily satisfy the distributivity property. As a result, every theorem of orthologic is a theorem of classical logic, but not conversely. This single omission enables orthologic to have much better computational properties than classical logic, while still being expressive enough to capture a large fragment of propositional reasoning tasks.

Canonical normalization in quadratic-time. We present a quadratic-time normalization algorithm that computes for any formula ϕ a normal form $\mathrm{NF}\,\phi$ such that $\mathrm{NF}\,\phi$ is a formula in the orthologic equivalence class of ϕ of smallest size (where size is measured as the number of conjunctions, disjunctions and literals in the formula's directed acyclic graph), and additionally $\phi \sim \psi \implies \mathrm{NF}\,\phi = \mathrm{NF}\,\psi$. Moreover, this algorithm preserves structure sharing, which is crucial at scale. As a preprocessing step, this normalization improves caching and reduces the number and cost of calls to external solvers. We implement and evaluate these techniques in the Stainless verifier [45], reducing solving time by up to 40% on a set of verification benchmarks.

Reasoning under non-logical assumptions. Real verification tasks rarely consist of closed formulas: reasoning takes place under assumptions (contexts) and domain axioms. However, a key limitation of orthologic, compared to classical logic, is that reasoning "under assumptions" cannot be reduced to validity by internalizing implication: because distributivity fails, the validity of $\neg\phi \vee \psi$ is not equivalent to asking whether ψ is valid whenever ϕ is valid. Hence, to solve the entailment problem (deciding equivalence of formulas under assumptions), we study the proof theory of orthologic.

Orthologic is characterized by an elegant restriction of the sequent calculus for classical logic **LK**: if we restrict **LK** to sequents containing at most two formulas, we obtain a sound and complete calculus for orthologic, which we call **LO**. We can then show that **LO** admits useful properties such as cut-elimination and the subformula property. These theorems generalize in the presence of non-logical axioms, allowing us to design an $\mathcal{O}(n^2(1 + |A|))$ algorithm for deciding the entailment problem under a finite set A

of non-logical axioms. The calculus **LO** then leads to a list of further useful results. For example, orthologic admits the interpolation property, it is sound and complete for various classes of formulas and it can be extended to support quantifiers.

Monotonic and antimonotonic function symbols in orthologic. In many applications, the language includes not only the logical connectives $\wedge, \vee, \neg, \bot, \top$, but also domain-specific connectives, quantifiers or uninterpreted operators. Often, some arguments are known to be monotonic or antimonotonic. For example, in first-order logic, both quantifiers are monotonic: if $\phi \leq \psi$ then $(\forall x.\ \phi) \leq (\forall x.\ \psi)$ and $(\exists x.\ \phi) \leq (\exists x.\ \psi)$. Hence, first-order formulas form an ortholattice where the order between formulas is provable implication and where $\forall x.\phi$ and $\exists x.\phi$ are monotonic function symbols (for every x). We generalize orthologic to support such function symbols (we call this extension OL^+) and extend the proof system **LO** to $\mathbf{LO}^+$ accordingly. The algorithms for normalization and the entailment problem also generalize to this setting while keeping the same asymptotic complexity of $\mathcal{O}(n^2)$ and $\mathcal{O}(n^2(1 + |A|))$ respectively.

A type-system application: subtyping with orthologic. A compelling application of OL^+ is subtyping with union and intersection types. In such type systems, the subtyping relation `<:` with union `|` and intersection `&` usually form a lattice. Additionally, covariant and contravariant type constructors (in particular, the constructor of function types $\Rightarrow$, which is contravariant in the first argument and covariant in the second argument) are exactly (anti)monotonic functions over this lattice. Existing implementations of such type systems are often incomplete, inefficient or rely on unintuitive assumptions compared to expected lattice behaviour.

We propose, instead, a simple and principled design: define subtyping judgements to hold if and only if they are provable from the laws of ortholattices together with the monotonicity and antimonotonicity of type constructors and the typing assumptions available in the context. This definition is intuitive, and efficiently decidable using our entailment algorithm for OL^+. In addition to unions and intersections, this approach naturally supports negation types, as well as common features such as nominal subtyping and record types.

Formal verification of orthologic-based reasoning. As components of a verification pipeline build up, so does the possibility of an implementation error. To ensure that they can be trusted, these verification algorithms should themselves be verified. Hence, we formalized key components of orthologic in the Rocq proof assistant: ortholattices with function symbols, the corresponding sequent calculus with the cut elimination theorem, a quadratic-time proof-search procedure and normalization for ortholattices and bounded lattices. Crucially, the proof search procedure relies on memoization and reference equality testing to achieve its optimal asymptotic complexity. From this formally verified implementation, we derive tactics that can be used to normalize or discharge

goals with any type and operations that form an ortholattice (and in particular the type `bool` of Booleans).

Even faster and weaker: orthocomplemented bisemilattices. Ortholattices are not the only candidates for a sound approximation of Boolean algebras. We study the class of *orthocomplemented bisemilattices* (*OCBSL*), which go even further and drop the absorption laws $a \wedge (a \vee b) = a$ and $a \vee (a \wedge b) = a$. In exchange, we obtain a normal-form algorithm running in quasilinear time $\mathcal{O}(n \log(n)^2)$. The resulting normal form is coarser than the orthologic normal form but still strictly stronger than negation-normal form. Here we do not rely on a proof system, but rather on the theory of term rewriting. This algorithm is also implemented in Stainless and can be optionally used instead of orthologic-based normalization. We provide a comparative evaluation of both algorithms.

Lisa

The most significant practical application of orthologic-based reasoning in this thesis is in the design of a new proof assistant called Lisa, based on set theory and first-order logic. Lisa implements orthologic normalization in its logical kernel and considers formulas equivalent in orthologic to be interchangeable. This makes proofs shorter and offers a predictable, efficient and trustworthy way to automatically prove the equivalence of propositional formulas under common rewrites, such as commutativity/associativity, constant folding and normalization. In practice, this improves usability, it is, in particular, a very useful component for more advanced tactics that require simplification (such as Lisa's 'Tautology' and 'Tableaux' tactics) or which often perform transformations in the orthologic fragment. In the second part of this thesis, we describe the theoretical foundations and implementation of Lisa.

Historical proof assistants have served and still serve well, and their established libraries hold great value. Yet at some point we need to explore new designs. This is necessary to try different — and potentially superior — approaches without being held back by old design decisions that are unchallenged because of compatibility, or simply out of habit.

The use of orthologic as a core reasoning engine, aiming to make proofs smaller and speed up library development, is one such design choice. Another choice is that of the foundations of the proof assistant. Most of the development effort in proof assistants has been carried out on systems based on some flavour of type theory. Experts in the field (for example, [48, 4, 74]) have argued that the prevalence of type theory as foundation of proof assistants in the last 30 years may owe more to historical contingency than to an innate superiority, and have advocated for the development of set-theory-based alternatives.

Design goals: the six virtues of modern proof systems. The design philosophy of Lisa is guided by six design goals:

- **Trust**: Proof systems should be built from small, well-understood logical foundations. Lisa is built on widely accepted and extensively studied mathematical foundations (First-Order logic and Set Theory) and has a small logical kernel.
- **Efficiency**: The proof system should be built on efficient (typically, polynomial-time) algorithms.
- **Predictability**: We should offer algorithms with clear completeness characterization when possible, since heuristics can be frustrating and hard to debug when they fail.
- **Usability**: It should be as easy as possible for humans and computers to use the system.
- **Interoperability**: We should make it as easy as possible for the system to be used by other systems and to export and import proofs to and from other systems.
- **Programmability**: As a computer system, a proof assistant should provide all the expressiveness allowed by a programming language.

Every particular design decision of Lisa comes down to balancing these six aspirations, and, when they conflict, finding the best possible balance between them. For example, the integration of orthologic-based reasoning is the result of a trade-off between trust, efficiency, predictability and usability.

Foundations: λFOL. First-order logic and ZFC set theory are the most widely accepted and studied foundations of mathematics among mathematicians. The theory has been stable and without any hint of inconsistency for more than a century, and successful tools such as Mizar, TLA$^+$ and Isabelle/ZF have demonstrated the practicality of such foundations for proof assistants.

Lisa builds on these foundations to maximize trust, but with some extensions to address some shortcomings of first-order logic for proof assistants. Concretely, we introduce λFOL, an extension of first-order logic formulated in the language of simply-typed lambda calculus, supporting higher-order syntax. Indeed, one of the key limitations of pure first-order logic is its inability to represent *terms* that bind variables, or contain subformulas, such as set comprehension expressions:

$$\{x \in A \mid P(x)\}$$

With λFOL, we aim to design a purely theoretical and logical extension that can be proved equivalent (for some definition of equivalence) to pure first-order logic. In contrast with approaches that rely on a stronger meta-logic to host first-order reasoning or which do not have a small logical kernel, λFOL is paired with a sequent-calculus-style deduction system closely aligned with traditional proof theory.

Implementation: a small kernel and a Scala-embedded proof language. Lisa is built according to the LCF architecture [35], with a small trusted *kernel* based on λFOL and orthologic that guarantees soundness, while a higher-level interface provides a richer syntax, automation and tooling. Some proof assistants have a dedicated proof script language but with limited programmability features. Others are written directly in the host language, which is better for automation but is typically hard to read and write. In contrast, Lisa's proof script language is implemented as a domain-specific language (DSL) embedded in Scala, which enables writing natural and readable proofs while still having the full expressivity of a modern programming language for writing tactics and automation. Proof steps in this DSL use a forward-style syntax (e.g., "have statement by tactic"), but remain ordinary executable code. This enables users to combine proof steps with regular programming constructs (loops, recursive functions, control flow operators) and the entire standard libraries of Scala and Java.

For the past three years, we have used Lisa to teach formal methods to graduate students. Moreover, it has been used by the teaching team of a programming class at EPFL (400 students) as an autograder and interface to teach undergraduate students how to formally reason about functional programs. The implementation approach of Lisa was key to making this project practically feasible.

Mechanizing higher-order logic. Even with set theory as foundation, everyday mathematics relies largely on typed reasoning and first-class functions. Types, usually under the name of "sets" or "collections" in vernacular mathematics, allow the resolution of overloaded names, automatic coercion of parameters and more generally help organize mathematical knowledge both for the machine and the human. This suggests viewing a large part of mathematics as a *soft type system over set theory*, where "types" are just sets.

In contrast, higher-order logic is a logic centred on simple types and first-class functions; it is expressive enough to formalize most mathematics, and it is the foundation of multiple successful proof assistants. To enable users to express and reason about typed statements and functions as in HOL, and to enable automated transfer of theorems from HOL libraries into Lisa, we formally embed HOL Light's higher-order logic into Lisa's set theory.

Interoperability of proof systems with SC-TPTP. Interoperability between proof systems, in particular interactive theorem provers (ITPs) and automated theorem provers (ATPs), has been a challenge for a long time. This is true even for systems with similar logical foundations, as they can have very different internal notions of proofs and granularity of proof steps, which need to be simulated when transferred between different systems.

Hence, we designed a proof format along with a toolchain for the transfer of proofs between proof systems based on first-order logic. This format is called SC-TPTP because its formalism is sequent calculus (SC), and it is an extension of the widely used TPTP

format for specifying problems for automated theorem provers. We developed a library of tools to parse, transform, check and print SC-TPTP proofs. On the ITP side, we developed extensions to make Lisa, HOL Light and Lean compatible with SC-TPTP. As for ATPs, we have implemented SC-TPTP support in Egg [99], Goéland [17] and Prover9 [61] (based on resolution). As a result, we obtain tactics that can be directly used in proof assistants. For example in Lisa, by using "*have* statement *by* {Egg | Goeland | Prover9}", Lisa invokes the external solver, which produces a proof and exports it into SC-TPTP, so that Lisa imports and verifies it.

1.2 Reader's guide

The remainder of this thesis is organized as follows.

Chapter 2 presents orthologic and its applications to automated and interactive reasoning.

Section 2.1: We introduce the necessary preliminaries on universal algebra, lattices and ortholattices.

Section 2.2: We present $\mathbf{LO}^+$, the sequent calculus for orthologic with axioms and (anti)monotonic function symbols, along with its main properties: soundness, completeness, cut-elimination, the subformula property and a quadratic-time proof search algorithm.

Section 2.3: We discuss more practical and optimized decision algorithms for particular fragments of the entailment problem in OL^+.

Section 2.4: We present the quadratic-time normal-form algorithm for OL^+. It first describes a normal-form algorithm for terms in the language of lattices with (anti)monotonic function symbols and then for ortholattices, with proofs of correctness and complexity.

Section 2.5: We show that orthologic admits the interpolation property and that interpolants are computable in quadratic time.

Section 2.6: We study the proof strength of orthologic with axioms and show that, under a natural encoding, it is complete for classes of formulas such as Horn. We then discuss an analogue of Tseitin transformation and relations to the resolution proof system.

Section 2.7: We extend orthologic to quantified orthologic and show soundness and completeness with respect to complete ortholattices along with the absence of quantifier elimination.

Section 2.8: We present a different extension of orthologic with predicates and variables that we call effectively propositional orthologic. We show that problems in ef-

fectively propositional orthologic with axioms can be decided in single-exponential time. The section then further discusses the expressivity of the class.

Section 2.9: We present an application of orthologic with (anti)monotonic function symbols to type systems. We define a simple programming language with union, intersection and negation types that uses orthologic entailment to decide subtyping judgements, along with a type checking algorithm. We then discuss additional features that can be represented in the system and shortcomings of existing approaches in widely used programming languages.

Section 2.10: We present the formalization of orthologic with (anti)monotonic function symbols in Rocq. The formalization encompasses the cut-elimination theorem, the correctness of the proof search algorithm and the correctness of orthologic normalization. The proof search procedure implements verified memoization and reference equality, which are crucial to achieve optimal complexity. This yields a set of tactics for Rocq that rely on this formalization.

Section 2.11: We answer the question of a fast and predictable weakening of classical logic by a different algebraic structure: orthocomplemented bisemilattices (*OCBSL*), which are weaker still than ortholattices. We present a quasilinear-time normal-form algorithm for this theory based on term rewriting, along with proofs of correctness and complexity.

Section 2.12: We present a set of benchmarks and experiments of some of the algorithms presented in this chapter. They compare the orthologic and *OCBSL* normal forms and show performance improvements in the Stainless verifier when using orthologic-based normalization. Finally, we show the relative performance of the Rocq tactics based on orthologic.

Chapter 3 discusses Lisa and interoperability of proof systems.

Section 3.1: We motivate and define λFOL, an extension of first-order logic using simply-typed lambda calculus, on which Lisa's logical kernel is based. The section also presents the corresponding proof system and discusses definition principles, schematic symbols and conservativity over traditional first-order logic.

Section 3.2: We illustrate how λFOL can be used in conjunction with set theory to conveniently express common mathematical notions such as set comprehension and function abstraction.

Section 3.3: We present the implementation of Lisa's logical kernel, based on λFOL and orthologic normalization. The section discusses Lisa's syntax, proof system and the mechanism by which new definitions, theorems and axioms are added to theories.

Section 3.4: We present Lisa's high-level user interface that builds on top of the kernel and the Scala-embedded DSL for type-checking expressions and writing proof scripts, and how to develop mathematical theories in Lisa. We then present the most important proof tactics and examples from Lisa's standard library.

Section 3.5: We present the formal embedding of HOL Light's higher-order logic in Lisa's set theory. This embedding represents HOL types and terms using λFOL, automated type checking and the replay of HOL Light's basic proof steps as Lisa tactics.

Section 3.6: We present the SC-TPTP format for exchange of proofs between first-order logic proof systems, its toolchain and the integration of SC-TPTP in Lisa, and other automated and interactive theorem provers.

Finally, Chapter 4 concludes this thesis and discusses possible directions for future work.

Dependencies between sections. The two chapters are independent of each other and can be read in either order. Within Chapter 2, Section 2.1 and Section 2.2 are prerequisites for all subsequent sections; Section 2.12 additionally requires Section 2.4 and Section 2.11. Within Chapter 3, sections follow a roughly linear order, but Section 3.6 can largely be read independently.

Typographic conventions. Code examples appear throughout this thesis in shaded blocks. The background color of each block indicates the language:

```
Scala / Lisa
```
```
Orthologic type system
```
```
Rocq
```
```
SC-TPTP
```

Reading aids. The appendix provides two reference aids: Section A.1 collects all notation and symbols used throughout the thesis, and Section A.2 lists all orthologic-related proof systems in full, with a diagram illustrating their relationships.

1.3 Published work, unpublished work and artefacts

This thesis contains a mix of published and unpublished work. The published work includes:

- Equivalence Checking for Orthocomplemented Bisemilattices in Log-Linear Time. Simon Guilloud, Viktor Kunčak. TACAS 2022. [39]
- Formula Normalizations in Verification. Simon Guilloud, Mario Bucev, Dragana Milovančević, Viktor Kunčak. CAV 2023. [42]
- LISA – A Modern Proof System. Simon Guilloud, Sankalp Gambhir, Viktor Kunčak. ITP 2023. [36]

- Interpolation and Quantifiers in Ortholattices. Simon Guilloud, Sankalp Gambhir, Viktor Kunčak. VMCAI 2024. [37]
- Orthologic With Axioms. Simon Guilloud, Viktor Kunčak. POPL 2024. [38]
- Mechanized HOL Reasoning in Set Theory. Simon Guilloud, Sankalp Gambhir, Andrea Gilot, Viktor Kunčak. ITP 2024. [43]
- Verified and Optimized Implementation of Orthologic Proof Search. Simon Guilloud, Clément Pit-Claudel. CAV 2025. [41]
- Interoperability of Proof Systems with SC-TPTP. Simon Guilloud, Julie Cailler, Sankalp Gambhir, Auguste Poiroux, Yann Herklotz, Thomas Bourgeat, Viktor Kunčak. CADE 30 (2025). [44]

Additionally, it contains contributions that have not been published elsewhere (as of January 2026), including:

- All extensions of orthologic with monotonic and antimonotonic function symbols (Section 2.2, Section 2.4).
- The application of orthologic-based reasoning to subtyping with union, intersection and negation types (Section 2.9).
- The improved algorithm for orthologic proof search (Subsection 2.3.1), which is the result of a collaboration with Vladislas de Haldat du Lys.
- The logical extension of first-order logic λFOL (Section 3.1) and its particular implementation in Lisa (Section 3.4).

The following artefacts are open source and available online:

- A repository containing the normalization algorithms for orthologic and orthocomplemented bisemilattices: *github.com/epfl-lara/lattices-algorithms*.
- The verified implementation of orthologic proof search and tactics in Rocq: *github.com/epfl-lara/rocl-orthologic*. It is also available as an opam package: *opam.ocaml.org/packages/orthologic-coq/*.
- The Lisa proof assistant: *github.com/epfl-lara/lisa*.
- The SC-TPTP toolchain: *github.com/SC-TPTP/sc-tptp/*.

1.4 Acknowledgement of contributions

The results presented in this thesis are the fruit of collaborations with many people over the years. I acknowledge the contribution of all my colleagues and coauthors — Mario Bucev, Dragana Milovančević, Sankalp Gambhir, Andrea Gilot, Clément Pit-Claudel,

Julie Cailler, Auguste Poiroux, Yann Herklotz, Thomas Bourgeat and Vladislas de Haldat du Lys — as well as my advisor Viktor Kunčak, for their contributions which ended up in this thesis. Specific contributions are detailed at the beginning of each section, when applicable. I also thank everyone, including many students, who contributed to the development of Lisa and hence indirectly to this thesis.

2 Orthologic

2.1 Background on universal algebra

We briefly review some necessary concepts of universal algebra. For a comprehensive introduction, we refer the reader to e.g. [86, Chapter 2].

Definition 2.1.1 (Algebras and signatures)**.** An *algebraic signature*, or simply *signature*, is a tuple of function symbols each annotated with an arity. For example, $(+^2, \times^2, -^1, 0^0, 1^0)$ is the signature of rings and fields. An *algebra* with signature $S = (f_1, \ldots, f_n)$ is a tuple $\mathcal{A} = (A, f_1^{\mathcal{A}}, \ldots, f_n^{\mathcal{A}})$ where A is a set, called the *universe* of $\mathcal{A}$ and each $f_i^{\mathcal{A}}$ is a function on A of arity equal to that of f_i. For example, a ring is a 6-tuple $(A, +^{\mathcal{A}}, \times^{\mathcal{A}}, -^{\mathcal{A}}, 0^{\mathcal{A}}, 1^{\mathcal{A}})$ where A is a set, $+^{\mathcal{A}}, \times^{\mathcal{A}}, -^{\mathcal{A}}$ are functions and $0^{\mathcal{A}}, 1^{\mathcal{A}}$ are elements of A, that is:

$$\begin{aligned} +^{\mathcal{A}} &: A \times A \to A \\ \times^{\mathcal{A}} &: A \times A \to A \\ -^{\mathcal{A}} &: A \to A \\ 0^{\mathcal{A}} &: A \\ 1^{\mathcal{A}} &: A \end{aligned}$$

In practice, we often omit the superscript $\mathcal{A}$ from functions. This technically creates an ambiguity between the function symbol and its corresponding function in the algebra, but the context usually makes it clear which one is meant.

The study of algebras themselves is rich, but we are specifically interested in syntactic expressions and statements about algebras, which allow us to ask, for example: "Are two given formulas always equivalent in Boolean algebras?" and "Is a given expression always equal to 0 in all rings?"

Definition 2.1.2 (Terms)**.** Let X be an infinite, countable set of variables. For an algebra signature S, let $\mathcal{T}_S(X)$ be the set of terms generated by X and S. Terms are trees whose leaves are labelled by members of X and constant symbols of S and nodes are labelled with function symbols of S of arity > 0. Formally,

$$\begin{aligned}\mathcal{T}_S(X) := \;& x \in X \\ & \mid c^0 \in S \\ & \mid f^n(t_1, \ldots, t_n) \text{ where } f^n \in S, t_i \in \mathcal{T}_S(X)\end{aligned}$$

We assume that all function symbols in S have unique identifiers, so that we can omit the arity part of function symbols without ambiguity.

For some binary function symbols $\cdot$, we sometimes use an infix notation and write $t_1 \cdot t_2$ instead of $\cdot(t_1, t_2)$. We denote by $\mathsf{FV}(t)$ the set of variables contained in a term t.

Definition 2.1.3. For a signature S and a term $t \in \mathcal{T}_S(X)$, we denote by $\|t\|$ the size of t, defined as the number of nodes in the tree representing t. If T is a set of terms, we define $\|T\| = \sum_{t \in T} \|t\|$. Note that this is in contrast with $|T|$, which denotes the cardinality of T.

Definition 2.1.4. An assignment is a function $\sigma : X \to \mathcal{T}_S(X)$ that maps variables to terms. For a term $t \in \mathcal{T}_S(X)$ and a variable $x \in X$, the *substitution* of an assignment σ in a term t, denoted $t[\sigma]$, is defined inductively as:

$$\begin{aligned} x[\sigma] &= \sigma(x) \\ f(t_1, \ldots, t_n)[\sigma] &= f(t_1[\sigma], \ldots, t_n[\sigma]) \end{aligned}$$

$t[x := s]$ is the substitution of the assignment σ such that $\sigma(x) = s$ and $\sigma(y) = y$ for all $y \neq x$. If T is a set of terms, we write $T[\sigma]$ for $\{t[\sigma] \mid t \in T\}$.

Example 2.1.5. Let $S_{\mathcal{R}} = (+^2, \times^2, -^1, 0^0, 1^0)$ be the signature of rings. Then $(x+1) \times (y + (-y))$ is a term in $\mathcal{T}_{S_{\mathcal{R}}}(X)$. In general, polynomials are terms over the signature of rings.

Let $S_{BA} = (\wedge^2, \vee^2, \neg^1, \bot^0, \top^0)$ be the signature of Boolean algebras. Then $(x \wedge y) \vee \neg z$ is a term in $\mathcal{T}_{S_{BA}}(X)$.

We often use an algebra or a class of algebras as implicitly standing for its signature. For example, we typically write $\mathcal{T}_{BA}(X)$ instead of $\mathcal{T}_{S_{BA}}(X)$.

Definition 2.1.6 (Interpretation)**.** For an algebra $\mathcal{A}$ with signature S and an assignment $\sigma : X \to A$, the *interpretation* of a term $t \in \mathcal{T}_S(X)$ in $\mathcal{A}$ under σ, denoted $[\![t]\!]_\sigma$ is defined inductively as:

$$\begin{aligned} [\![x]\!]_\sigma &= \sigma(x) \text{ for } x \in X \\ [\![c^0]\!]_\sigma &= c^{\mathcal{A}} \\ [\![f(t_1, \ldots, t_n)]\!]_\sigma &= f^{\mathcal{A}}([\![t_1]\!]_\sigma, \ldots, [\![t_n]\!]_\sigma) \end{aligned}$$

For two terms $s, t \in \mathcal{T}_S(X)$, we say that s and t are *equivalent* in $\mathcal{A}$, denoted $s \sim_{\mathcal{A}} t$, if for all assignments σ, $[\![s]\!]_\sigma = [\![t]\!]_\sigma$. We also say that $s = t$ *holds* in $\mathcal{A}$ or that $\mathcal{A}$ is a model of $s = t$.

Definition 2.1.7 (Algebraic classes and varieties)**.** A *class* of algebras is a collection of algebras that all share the same signature. For example, we have the class of all rings, the class of all Boolean algebras, or the class of all lattices. If K is a class of algebras, we write $s \sim_K t$ if $s \sim_{\mathcal{A}} t$ for all $\mathcal{A} \in K$.

A *variety* V is the class of all algebras with signature S defined by a set of universally quantified identities of the form $s_i = t_i$, where $s_i, t_i \in \mathcal{T}_S(X)$. Formally, for any algebra $\mathcal{A}$ with signature S, $\mathcal{A} \in V$ if and only if for all i and σ,

$$s_i \sim_{\mathcal{A}} t_i$$

Example 2.1.8. Groups, Abelian groups, rings, fields, lattices, Boolean algebras and ortholattices are all varieties. For example, the class of all Abelian groups is the variety defined by the signature $(\cdot^2, {}^{-1^1}, e^0)$ and the identities:

$$\begin{aligned}(x \cdot y) \cdot z &= x \cdot (y \cdot z)\\ x \cdot y &= y \cdot x\\ e \cdot x &= x\\ x \cdot x^{-1} &= e\end{aligned}$$

All the classes of algebras we are interested in are varieties. Varieties have an entirely different characterization in terms of closure properties, and that does not use the notion of term: A class of algebras is a variety if and only if it is closed under homomorphic images, subalgebras and direct products. The equivalence of these characterizations is called Birkhoff's theorem [86, Theorem 11.9].

Definition 2.1.9 (Congruences and quotient algebras)**.** A relation $\sim$ on a set A is an *equivalence* relation if it is reflexive, symmetric and transitive, that is:

$$\begin{aligned}\forall a \in A.\ & a \sim a\\ \forall a, b \in A.\ & a \sim b \implies b \sim a\\ \forall a, b, c \in A.\ & a \sim b \implies b \sim c \implies a \sim c\end{aligned}$$

A relation is a *congruence* relation with respect to an algebra $\mathcal{A} = (A, f_1, \ldots, f_n)$ if it is an equivalence relation on A and, for each function symbol f_i of arity k, for all $a_1, \ldots, a_k, b_1, \ldots, b_k \in A$ such that $a_j \sim b_j$, we have

$$f_i(a_1, \ldots, a_k) \sim f_i(b_1, \ldots, b_k)$$

The *quotient algebra* $\mathcal{A}/\!\sim$ is the algebra $(A/\!\sim, f_1', \ldots, f_n')$ where $A/\!\sim$ is the set of equivalence classes of $\sim$ and for each function symbol f_i of arity k, f_i' is defined as:

$$f_i'([a_1]_\sim, \ldots, [a_k]_\sim) = [f_i(a_1, \ldots, a_k)]_\sim$$

Thanks to the congruence property, f_i' is well-defined, that is, it does not depend on the choice of representatives a_j of their equivalence classes.

Theorem 2.1.10 (Quotient variety, [86, Theorem 9.5])**.** Let $\mathcal{A}$ be an algebra in a variety V and $\sim$ a congruence relation on $\mathcal{A}$. Then the quotient algebra $\mathcal{A}/\sim$ is a member of V.

For every variety there is an algebra of particular interest that is the most general algebra in the variety, called the *free algebra* of the variety.

Definition 2.1.11 (Term algebra and free algebra)**.** Let S be an algebraic signature. The *term algebra* of S is the algebra with base set $\mathcal{T}_S(X)$ and where for each function symbol $f^n \in S$, the corresponding function $f : \mathcal{T}_S(X)^n \to \mathcal{T}_S(X)$ is defined as

$$f(t_1, \ldots, t_n) = f^n(t_1, \ldots, t_n)$$

We use $\mathcal{T}_S(X)$ both to denote the set of terms with signature S and the corresponding term algebra.

Let V be a variety with signature S_V. For a set of variables X, the *free algebra* $\mathcal{F}_V(X)$ of a variety V is the quotient algebra $\mathcal{T}_{S_V}(X)/\sim_V$.

Note that by definition, for any two terms $s, t \in \mathcal{T}_S(X)$, $s \sim_V t$ if and only if $[s]_{\sim_V} = [t]_{\sim_V}$. Hence $\mathcal{F}_V(X) \in V$. In fact, $\mathcal{F}_V(X)$ is the most general member of V in the following sense: an equality $s = t$ holds in $\mathcal{F}_V(X)$ if and only if it holds in all algebras of V. Moreover, using Gödel's completeness theorem for first-order logic, $s = t$ holds in all members of V if and only if it is provable (in first-order logic) from the axioms of V.

In fact, a result, again from Birkhoff, states that we do not need all of first-order logic, but only the weaker *equational logic*, which only allows reasoning with substitution of equal terms and instantiation of free variables.

Theorem 2.1.12 (Completeness of equational logic [86, Theorem 14.19])**.** Let V be a variety with axioms $s_1 = t_1, \ldots, s_n = t_n$. For any two terms $s, t \in \mathcal{T}_V(X)$, the following are equivalent:

- $s \sim_V t$
- $s = t$ holds in $\mathcal{F}_V(X)$
- $s = t$ is provable from $s_1 = t_1, \ldots, s_n = t_n$ in first-order logic with equality.
- $s = t$ is provable from $s_1 = t_1, \ldots, s_n = t_n$ in equational logic.

Definition 2.1.13. A ground term is a term that contains no variables. Similarly, a ground identity is an identity between two ground terms.

Definition 2.1.14 (Word problem)**.** The *word problem* for a fixed class of algebras K consists in deciding, for any pair $s, t \in \mathcal{T}_K(X)$, whether $s \sim_K t$.

A (finite) presentation for K is a (finite) set of ground identities. If $A = \{(s_1, t_1), \ldots, (s_n, t_n)\}$ is a presentation, we denote $K|A$ the restriction of K to algebras satisfying $s_1 = t_1, \ldots, s_n = t_n$.

The *entailment problem* (also called uniform generalized word problem) consists in deciding, for any given finite presentation $A = \{(s_1, t_1), \ldots, (s_n, t_n)\}$ and pair of terms s and t, if $s \sim_{K|A} t$.

In the terminology of classical first-order logic, the laws of a variety are a finite set of *universally quantified* formulas, $\mathcal{T}$, and they can be instantiated. The presentation A is a set of *quantifier-free* formulas. If T is the set of universally quantified statements characterizing a variety, we can then view the uniform word problem as a special case of the question of consequence in first-order logic:

$$\mathcal{T} \cup A \vdash s = t$$

2.1.1 Lattices and ortholattices

Ortholattices form an algebra with the same signature as Boolean algebras, but with a weaker structure. In particular, the distributivity law does not necessarily hold. Table 2.1 shows their axiomatization. All Boolean algebras are ortholattices; they are precisely those ortholattices that are distributive. Figure 2.1 shows two characteristic finite non-distributive ortholattices; keeping these structures in mind may provide intuition for reasoning inside the class of all ortholattices. *Orthologic* is the logical system that corresponds to ortholattices, analogously to how classical logic corresponds to Boolean algebras and intuitionistic logic to Heyting algebras. Formally, ortholattices are the Lindenbaum-Tarski algebras of orthologic.

V1:	$x \vee y = y \vee x$	V1':	$x \wedge y = y \wedge x$
V2:	$x \vee (y \vee z) = (x \vee y) \vee z$	V2':	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
V3:	$x \vee x = x$	V3':	$x \wedge x = x$
V4:	$x \vee (x \wedge y) = x$	V4':	$x \wedge (x \vee y) = x$
V5:	$x \vee \top = \top$	V5':	$x \wedge \bot = \bot$
V6:	$x \vee \bot = x$	V6':	$x \wedge \top = x$
V7:	$\neg\neg x = x$		
V8:	$x \vee \neg x = \top$	V8':	$x \wedge \neg x = \bot$
V9:	$\neg(x \vee y) = \neg x \wedge \neg y$	V9':	$\neg(x \wedge y) = \neg x \vee \neg y$

Table 2.1: Laws of ortholattices, algebraic varieties with signature $(\wedge, \vee, \bot, \top, \neg)$.

Definition 2.1.15 (Lattices and ortholattices)**.** A *lattice* is an algebraic structure with signature $(A, \wedge^2, \vee^2)$ satisfying laws V1-V4 and V1'-V4' of Table 2.1.

V10: $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	V10': $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$

Table 2.2: Distributivity laws of Boolean algebras, which do not necessarily hold in ortholattices.

A *bounded lattice* is an algebraic structure with signature $(A, \wedge^2, \vee^2, \bot^0, \top^0)$ which satisfies laws V1-V6 and V1'-V6' of Table 2.1. $\bot$ is the smallest element of the lattice, and $\top$ the largest.

An *ortholattice* is an algebraic structure with signature $(A, \wedge^2, \vee^2, \neg^1, \top^0, \bot^0)$ which satisfies the laws V1-V9 and V1'-V9' of Table 2.1.

A *Boolean algebra* is an ortholattice that also satisfies the distributivity laws V10 and V10' of Table 2.2.

We use L, BL, OL, BA to denote the varieties of, respectively, all lattices, bounded lattices, ortholattices and Boolean algebras.

Since elements of $\mathcal{T}_L(X), \mathcal{T}_{BL}(X), \mathcal{T}_{OL}(X)$ and $\mathcal{T}_{BA}(X)$ correspond to formulas of propositional logic, we often refer to them as formulas instead of terms.

In any lattice, we can define a natural order relation.

Definition 2.1.16. A *partially ordered set* (or *poset*) is a set S equipped with a binary relation $\leq$ that is reflexive, antisymmetric and transitive, that is, for all $x, y, z \in S$:

$$\begin{aligned} &x \leq x && \text{(reflexivity)} \\ &x \leq y \wedge y \leq x \implies x = y && \text{(antisymmetry)} \\ &x \leq y \wedge y \leq z \implies x \leq z && \text{(transitivity)} \end{aligned}$$

Definition 2.1.17. For any lattice $(S, \wedge, \vee)$, define a binary relation $\leq$ on S as

$$a \leq b \iff (a = (a \wedge b))$$

This is also equivalent to $(b = (a \vee b))$. Indeed, assume $a = (a \wedge b)$.

$$\begin{aligned} a &= a \wedge b \\ b \vee a &= b \vee (a \wedge b) && \text{by congruence} \\ b \vee a &= b && \text{by law V4} \\ a \vee b &= b && \text{by law V1} \end{aligned}$$

With the relation $\leq$ defined as above, every lattice is also a poset that satisfies the laws shown in Table 2.3 [55, 66]. The converse is also true:

Theorem 2.1.18. Let $(S, \leq)$ be a poset. If there are binary operations $\wedge$ and $\vee$ satisfying laws P1-P3, P1'-P3' of Table 2.3, then $(S, \wedge, \vee)$ is a lattice.

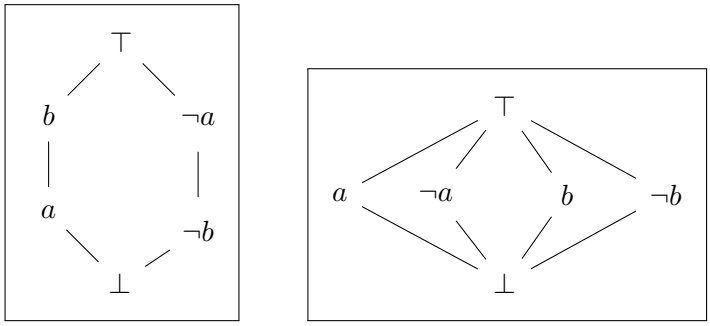


Figure 2.1: Ortholattices O_6 and M_4. An ortholattice is distributive if and only if it contains neither as a subortholattice.

If there are also constants $\bot$ and $\top$ satisfying laws P4, P4' of Table 2.3, then $(S, \wedge, \vee, \bot, \top)$ is a bounded lattice.

If there is also a unary operation $\neg$ satisfying laws P5-P7, P5'-P7' of Table 2.3, then $(S, \wedge, \vee, \bot, \top, \neg)$ is an ortholattice.

P1:	$x \wedge y \leq x$	P1':	$x \leq x \vee y$
P2:	$x \wedge y \leq y$	P2':	$y \leq x \vee y$
P3:	$x \leq y \ \& \ x \leq z \implies x \leq y \wedge z$	P3':	$x \leq z \ \& \ y \leq z \implies x \vee y \leq z$
P4:	$\bot \leq x$	P4':	$x \leq \top$
P5:	$x \leq \neg\neg x$	P5':	$\neg\neg x \leq x$
P6:	$x \leq y \implies \neg y \leq \neg x$		
P7:	$x \wedge \neg x \leq \bot$	P7':	$\top \leq x \vee \neg x$

Table 2.3: Axiomatization of ortholattices in the signature $(\leq, \wedge, \vee, \bot, \top, \neg)$ as partially ordered sets. We use & for conjunction between atomic formulas of axioms, to differentiate it from the term-level lattice operation $\wedge$. This axiomatization is equivalent to the one in Table 2.1.

By definition, Boolean algebras are precisely ortholattices that are distributive. Figure 2.1 shows ortholattices O_6 and M_4 that are *not* Boolean algebras. In fact, an ortholattice is a Boolean algebra if and only if it contains neither O_6 nor M_4 as a subortholattice [25, 55].

Note that the laws of both *OL* and *BA* imply that $\bot$ can always be represented by $x \wedge \neg x$ and $\top$ as $x \vee \neg x$. To simplify proofs, we thus will often omit the cases corresponding to $\bot$ and $\top$.

Monotonic function symbols in lattices

In many applications, our domain will be stronger and richer than simply a lattice or ortholattice. It may admit additional laws, but also additional function symbols beyond $\wedge, \vee$ and $\neg$. Moreover, in many cases these additional function symbols will be known to be monotonic in their arguments.

Definition 2.1.19 (Monotonic functions)**.** L^+ (resp. BL^+, OL^+) is the class whose algebras are lattices (resp. bounded lattices, ortholattices) with countably many additional function symbols $F^{i,j,k}$, (for $i, j, k \in \mathbb{N}$ and F in some infinite alphabet), each of arity $i + j + k$, satisfying for each such symbol the law V11 of Table 2.4. We say that $F^{i,j,k}$ is *invariant* in the first i arguments, *covariant* (or monotonic) in the next j arguments and *contravariant* (or antimonotonic) in the last k arguments.

$$\begin{aligned} \text{V11:} \qquad & \forall n \in \{1, \dots, j\}.\ y_n \leq y'_n \implies \\ & \forall n \in \{1, \dots, k\}.\ z'_n \leq z_n \implies \\ & F^{i,j,k}(x_1, \dots, y_1, \dots, z_1, ...) \leq F^{i,j,k}(x_1, \dots, y'_1, \dots, z'_1, ...) \end{aligned}$$

Table 2.4: Monotonicity law for (ortho)lattices with function symbols.

Example 2.1.20. The set of (equivalence classes of provably-equivalent) first-order logic formulas is an OL^+, where quantifiers $\forall$ and $\exists$ are unary function symbols that are covariant, and $\phi \leq \psi$ is provability of $\phi \implies \psi$. Indeed, for any formula ϕ and ψ and variable x, we have

$$(\phi \leq \psi) \implies (\forall x.\ \phi \leq \forall x.\ \psi) \text{ is provable}$$

A type system with subtyping, union (corresponding to $\vee$) and intersection (corresponding to $\wedge$) types and smallest and largest types can naturally be seen as a bounded lattice. Type constructors such as `List` then correspond to function symbols, making the type system a BL^+. Examples of programming languages whose type systems are a BL^+ include Scala, TypeScript and Flow. Additionally, if the type system supports negation types, then it is an OL^+. We will further study OL^+-based type systems in Section 2.9.

Laws V11 and V11' of Table 2.4 are not simple identities, but rather *quasi-identities*, that is, implications over identities. Hence, L^+, BL^+ and OL^+ are not a priori varieties, but we can show that they are.

Theorem 2.1.21. L^+, BL^+ and OL^+ are varieties.

Proof. For every symbol F, axiom V11 is provably equivalent to

$$F^{i,j,k}(x_1, \dots, y_1, \dots, z_1, ...) \leq F^{i,j,k}(x_1, \dots, y_1 \vee y'_1, \dots, z_1 \wedge z'_1, ...)$$

In every lattice:

- V11 $\implies$ V11'. In $V11$, instantiate y'_j by $y_j \vee y'_j$ and z'_k by $z_k \wedge z'_k$. Since $y_j \leq_L y_j \vee y'_j$ and $z'_k \leq_L z_k \wedge z'_k$, the conditions of V10 are fulfilled and we conclude.
- V11' $\implies$ V11. Since by assumption $y_j \leq y'_j$, we have $y_j \vee y'_j = y'_j$. Similarly, since by assumption $z'_j \leq z_j$, we have $z_j \wedge z'_j = z'_j$, so we conclude.

Hence the classes L^+, BL^+ and OL^+ are characterized by the axioms of, respectively, lattices, bounded lattices and ortholattices as well as V11', and hence are varieties. □

Word problem in lattices and ortholattices. Let A be a lattice. Then for any s, t in $\mathcal{T}_L(X)$, $s \sim_A t$ if and only if $s \leq_A t$ and $t \leq_A s$. Hence, the word problem for lattices reduces to deciding whether $s \leq_L t$.

Similarly, we can relax the definition of a presentation for lattices to equivalently be a set of inequalities rather than equalities. Indeed, an inequality $s_i \leq_A t_i$ is equivalent to the equality $s_i = s_i \wedge t_i$.

Note on notation for proof systems. In the rest of this thesis, we will define many slightly different proof systems, with names which, along with names for the corresponding classes of algebras, can be confusing. For clarity, proof systems will always be denoted in bold, straight font, e.g. **LO**, while algebras will be denoted in italics, e.g. *OL*. An exhaustive list of all proof systems used in this chapter, along with their proof steps and a summarizing diagram is given in Section A.2.

2.2 Orthologic proof system

Sequent calculi form a family of proof systems which have been instrumental to the development of proof theory, providing a framework for analysing the structure of logical derivations. They were originally introduced in two variants, LK for classical logic and LJ for intuitionistic logic, in the 1930s. In a sequent calculus, statements are *sequents*, which consist of a pair of sets of formulas:

$$\phi_1, \phi_2, \ldots \vdash \psi_1, \psi_2, \ldots$$

The formulas on the left side of $\vdash$ are called antecedents and those on the right succedents. The original sequent calculi LJ and LK have been shown to admit several important properties and in particular the *cut elimination* theorem, which in turn implies the *interpolation property*, the *subformula property* and the consistency of classical and intuitionistic first-order logic. Later on, sequent calculi have been developed for other logics, such as linear logic (LL). The significance of these proof systems and what makes them particularly suitable for analysis comes from the natural structure and symmetry in their logical inference rules. A somewhat remarkable fact is that both LJ and LL are characterized by syntactic restrictions on the sequents that appear in the proofs, while having essentially the same inference rules and axioms as the more powerful calculus LK for classical logic: in LJ there can only be at most one succedent formula, while in LL antecedents and succedents are ordered lists rather than sets. See Table 2.5 for a comparison of sequent calculi for different logics with a lattice structure (i.e. with $\wedge$ and $\vee$). Beyond proof theory, sequent calculi have applications in automated deduction, and even in programming languages and software verification.

It is hence not overly surprising that there exists a sequent calculus for orthologic. On the other hand, it is quite remarkable that the particular syntactic restriction from LK that characterizes orthologic is so simple and key to the good complexity of orthologic-related proof systems.

2.2.1 Sequent calculus for orthologic

As sequents in orthologic can only contain at most two formulas, it is convenient to represent them as a pair of annotated formulas.

Definition 2.2.1 (Annotated formulas, sequents)**.** An *annotated formula* is either ϕ^L or ϕ^R, where ϕ is a formula over the language $\vee, \wedge, \neg, \bot, \top$ of orthologic. We denote arbitrary annotated formulas as Γ, Δ. A sequent is an unordered pair of annotated formulas (Γ, Δ), often written without parentheses.

Definition 2.2.2. The sequent calculus for orthologic **LO** contains the rules defined in Table 2.6.

A sequent Γ, Δ is *provable* in **LO** if there exists a proof of it. Additionally, we may admit an arbitrary set of non-logical axioms A, containing sequents that are assumed to

Name	Logic	Algebra	Restrictions	Remark
LK	Classical logic	Boolean algebra	No restriction	Has proper $\neg$ $\Longrightarrow$, $\top$, $\bot$
LJ	Intuitionistic logic[a]	Heyting algebra	At most one succedent	Has a proper $\Longrightarrow$, but not a 'true' $\neg$
	Minimal logic	Unbounded Heyting algebra	Exactly one succedent	Has a proper $\Longrightarrow$, but no $\neg$ nor $\bot$
LO *	Orthologic	Ortholattice	At most two formulas	Has a proper $\neg$, but not a 'true' $\Longrightarrow$
LL	Linear logic[b]	Girale [2]	Sequents are pairs of lists rather than sets	Assumptions cannot be used multiple times
$\mathbf{LO}_{\boldsymbol{L}}$ *		Bounded Lattices	At most one succedent and one antecedent	No $\neg$ nor $\Longrightarrow$. At most one assumption
$\mathbf{LO}_{\boldsymbol{BL}}$ *		Lattices	Exactly one succedent and one antecedent	Neither $\neg$, $\top$, $\bot$ nor $\Longrightarrow$

Table 2.5: Comparison of sequent calculi for different logics. *: Names introduced in the present work.

hold and can be used as leaves in a proof:

$$\frac{}{\Gamma, \Delta}\ \textsc{Axiom} \quad \text{if } (\Gamma, \Delta) \in A$$

If a sequent Γ, Δ is provable in **LO**, we write $\vdash_{\mathbf{LO}} \Gamma, \Delta$.

Throughout this section, we assume an arbitrary fixed set of axioms A, and every theorem and definition is implicitly parametric on A.

The Replace rule is equivalent to the more typical *weakening* rule in LJ and LK, which allows one to add arbitrary formulas to either side of a sequent. As a sequent with a single formula Γ on either side is equivalent to the sequent Γ, Γ containing it twice, and since in orthologic, all sequents have at most two formulas, it is more convenient in implementations to say that every sequent has *exactly* two formulas.

To support reasoning with possibly (anti)monotonic function symbols in OL^+, we extend **LO** to $\mathbf{LO}^+$.

Definition 2.2.3. The proof system $\mathbf{LO}^+$ for sequents over the language of ortholattices with function symbols (Definition 2.1.19) extends **LO** with the following rule for every function symbol F with i invariant, j monotonic and k antimonotonic parameters, called the *F-rule*:

$$\frac{\phi^L_{x_1}, \psi^R_{x_1} \qquad \psi^L_{x_1}, \phi^R_{x_1} \quad ... \qquad \phi^L_{y_1}, \psi^R_{y_1} \quad ... \qquad \psi^L_{z_1}, \phi^R_{z_1} \quad ...}{F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)^L, F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)^R}\ \text{F-RULE}$$

$$\frac{}{\phi^L, \phi^R}\ \textsc{Hyp}$$

$$\frac{\Gamma, \psi^R \qquad \psi^L, \Delta}{\Gamma, \Delta}\ \textsc{Cut}$$

$$\frac{\Gamma, \Gamma}{\Gamma, \Delta}\ \textsc{Replace}$$

$$\frac{}{\bot^L, \Delta}\ \textsc{LeftBot} \qquad \frac{}{\Gamma, \top^R}\ \textsc{RightTop}$$

$$\frac{\Gamma, \phi^L}{\Gamma, (\phi \wedge \psi)^L}\ \textsc{LeftAnd} \qquad \frac{\Gamma, \phi^R \qquad \Gamma, \psi^R}{\Gamma, (\phi \wedge \psi)^R}\ \textsc{RightAnd}$$

$$\frac{\Gamma, \phi^L \qquad \Gamma, \psi^L}{\Gamma, (\phi \vee \psi)^L}\ \textsc{LeftOr} \qquad \frac{\Gamma, \phi^R}{\Gamma, (\phi \vee \psi)^R}\ \textsc{RightOr}$$

$$\frac{\Gamma, \phi^R}{\Gamma, (\neg\phi)^L}\ \textsc{LeftNot} \qquad \frac{\Gamma, \phi^L}{\Gamma, (\neg\phi)^R}\ \textsc{RightNot}$$

$$\frac{}{\Gamma, \Delta}\ \textsc{Axiom} \quad \text{if } (\Gamma, \Delta) \in A$$

Table 2.6: Proof system **LO** for ortholattices.

If a sequent Γ, Δ is provable in $\mathbf{LO}^+$, we write $\vdash_{\mathbf{LO}^+} \Gamma, \Delta$.

Throughout this section, we assume an arbitrary fixed set of axioms A and function symbols. Note that OL is the particular case of OL^+ where there are no function symbols. Hence, all results about OL^+ and $\mathbf{LO}^+$ also hold for OL and **LO**.

Definition 2.2.4 (Interpretation of sequents)**.** Given $\mathcal{O} \in OL^+$ an ortholattice with function symbols, a sequent s and an assignment $\alpha : \mathcal{V} \to \mathcal{O}$ where $\mathcal{V}$ is the fixed set of variables, define $[\![s]\!]_\alpha$ as:

$$[\![\phi^L, \psi^R]\!]_\alpha = [\![\phi]\!]_\alpha \leq [\![\psi]\!]_\alpha$$
$$[\![\phi^L, \psi^L]\!]_\alpha = [\![\phi]\!]_\alpha \leq [\![\neg\psi]\!]_\alpha$$
$$[\![\phi^R, \psi^R]\!]_\alpha = [\![\neg\phi]\!]_\alpha \leq [\![\psi]\!]_\alpha$$

We say that an ortholattice satisfies a sequent s if for every assignment α, $[\![s]\!]_\alpha$ holds.

Theorem 2.2.5 (Soundness of OL^+)**.** Let s be a sequent. If $\vdash_{\mathbf{LO}^+} s$ then every ortholattice that satisfies all the axioms in A also satisfies s.

Proof. Fix the ortholattice and assignment, and proceed by induction on the proof of s. For every rule in $\mathbf{LO}^+$, if the premises of the rule hold, then the conclusion also holds. We show the case of LeftAnd as an example.

For any assignment $\sigma : X \to \mathcal{O}$, the interpretation of the conclusion of a LeftAnd rule is

$$\llbracket \Gamma, (\phi \wedge \psi)^L \rrbracket_\sigma$$

Γ can be empty, a left formula or a right formula. If it is empty then

$$\llbracket \Gamma, (\phi \wedge \psi)^L \rrbracket_\sigma \iff \llbracket \bot^R, (\phi \wedge \psi)^L \rrbracket_\sigma$$

If $\Gamma = \gamma^L$, then we have

$$\llbracket \Gamma, (\phi \wedge \psi)^L \rrbracket_\sigma \iff \llbracket (\neg\gamma)^R, (\phi \wedge \psi)^L \rrbracket_\sigma .$$

So without loss of generality we can assume $\Gamma = \gamma^R$ to be a right formula. Now,

$$\begin{aligned} \llbracket \gamma^R, (\phi \wedge \psi)^L \rrbracket_\sigma &\iff \\ \llbracket \phi \wedge \psi \rrbracket_\sigma \leq \llbracket \gamma \rrbracket_\sigma &\iff \\ \llbracket \phi \rrbracket_\sigma \wedge \llbracket \psi \rrbracket_\sigma \leq \llbracket \gamma \rrbracket_\sigma & \end{aligned}$$

But using the premise of the LeftAnd rule and the induction hypothesis, we know $\llbracket \gamma^R, \phi^L \rrbracket_\sigma$ holds. Hence,

$$\begin{aligned} \llbracket \gamma^R, \phi^L \rrbracket_\sigma &\iff \\ \llbracket \phi \rrbracket_\sigma \leq \llbracket \gamma \rrbracket_\sigma &\implies \\ \llbracket \phi \rrbracket_\sigma \wedge \llbracket \psi \rrbracket_\sigma \leq \llbracket \gamma \rrbracket_\sigma & \end{aligned}$$

where the implication holds by the laws of ortholattices (Table 2.3). □

Theorem 2.2.6 (Completeness of $\mathbf{LO}^+$)**.** Let s be a sequent. If s holds in every ortholattice that satisfies the axioms in A, then $\vdash_{\mathbf{LO}^+} s$.

Proof. For $\phi, \psi \in \mathcal{T}_{OL^+}(X)$, let $\phi \dashv\vdash \psi$ be the relation defined by

$$\phi \dashv\vdash \psi \iff (\vdash_{\mathbf{LO}^+} \phi^L, \psi^R) \text{ and } (\vdash_{\mathbf{LO}^+} \psi^L, \phi^R)$$

It is not difficult to prove that $\dashv\vdash$ is a congruence relation with respect to $\wedge$, $\vee$ and $\neg$, and that $\mathcal{T}_{OL^+}(X)_{/\dashv\vdash}$ is itself an ortholattice with $\phi_{/\dashv\vdash} \leq \psi_{/\dashv\vdash}$ defined as $\vdash_{\mathbf{LO}^+} \phi^L, \psi^R$. Hence, if $\phi \leq \psi$ holds in all ortholattices, it holds in particular in $\mathcal{T}_{OL^+}(X)_{/\dashv\vdash}$, and hence by definition $\vdash_{\mathbf{LO}^+} \phi^L, \psi^R$. □

Soundness and completeness give us an alternative characterization of the generalized word problem for OL^+: an inequality $\phi \leq \psi$ is true in all ortholattices if and only if it is provable in $\mathbf{LO}^+$.

Towards a proof search procedure for $\mathbf{LO}^+$, note that all rules except for the Cut rule have the subformula property: all formulas appearing in the premises of a rule are subformulas of the formulas appearing in the conclusion. We aim to use this property

to guide proof search, but we first need to show that the Cut rule can be suitably restricted.

The following corollaries to completeness give us useful rules:

Corollary 2.2.7. $\mathbf{LO}^+$ admits the following proof step:

$$\frac{\Gamma[x := \phi], \Delta[x := \phi] \qquad \phi^L, \psi^R \qquad \phi^R, \psi^L}{\Gamma[x := \psi], \Delta[x := \psi]} \textsc{Subst}$$

Proof. If ϕ^L, ψ^R and ϕ^R, ψ^L are provable, then in any ortholattice and for any σ, $[\![\phi]\!]_\sigma = [\![\psi]\!]_\sigma$ and hence

$$[\![\Gamma[x := \phi], \Delta[x := \phi]]\!]_\sigma = [\![\Gamma[x := \psi], \Delta[x := \psi]]\!]_\sigma .$$

Hence by completeness, there always exists a proof of $\Gamma[x := \psi], \Delta[x := \psi]$. □

Lemma 2.2.8 (Instantiation)**.** A formula G is covariant (respectively contravariant) in a variable x if for any two formulas ϕ and ψ such that $\vdash_{\mathbf{LO}^+} \phi^L, \psi^R$ (respectively $\vdash_{\mathbf{LO}^+} \psi^L, \phi^R$), it holds that

$$\vdash_{\mathbf{LO}^+} G[x := \phi]^L, G[x := \psi]^R$$

Let $F(A_1, \ldots, A_n)$ be a function symbol of arity n, and let $G_{A_1,\ldots,A_n}$ be an arbitrary formula such that for every A_i, if F is covariant (respectively contravariant) in A_i then G is covariant (respectively contravariant) in A_i. Then for any set of axioms C and annotated formulas Γ and Δ such that

$$\vdash_C \Gamma, \Delta$$

We have

$$\vdash_{C[F:=G]} \Gamma[F := G], \Delta[F := G]$$

Proof. Follows from soundness and completeness. Since $\vdash_C \Gamma, \Delta$ is provable, it necessarily holds in every OL^+ with any valuation of function symbols and variables that satisfies axioms in C. One particular such OL^+ is $\mathcal{T}_{OL^+}$ with the valuation map that is the identity on every variable and every function symbol but F, mapping F to the function that maps $x_1, \ldots, x_n$ to $G[A_1 := x_1, \ldots, A_n := x_n]$. Note that we require the term G to have the same variance as F, as otherwise it would not satisfy V10. □

Definition 2.2.9 ($\mathbf{CF}^+$)**.** The proof system $\mathbf{CF}^+$ contains the rules of $\mathbf{LO}^+$ except for Axiom and Cut, and adds instead the rule

$$\frac{\Gamma, \phi^R \qquad \psi^L, \Delta}{\Gamma, \Delta} \textsc{AxiomCut} \quad (\text{with } (\phi, \psi) \in A)$$

Note that if $A = \emptyset$, then AxiomCut cannot be instantiated and in this case $\mathbf{CF}^+$ is precisely $\mathbf{LO}^+$ without Cut.

2.2.2 Cut elimination for $\mathbf{LO}^+$

We now show that $\mathbf{LO}^+$ admits a form of cut elimination, in the sense that every sequent provable in $\mathbf{CF}^+$ is provable in $\mathbf{LO}^+$ without the Cut rule. This is a crucial restriction for efficient proof search.

Theorem 2.2.10 (Partial Cut elimination)**.** A sequent has a proof in $\mathbf{LO}^+$ if and only if it has a proof in $\mathbf{CF}^+$.

Proof. For the 'if' direction, it suffices to note that AxiomCut is indeed a restriction of the Axiom and Cut rules. For the 'only if' direction, first note that the original form of the Axiom rule can be easily recovered with Hyp, as

$$\frac{}{\phi^L, \psi^R}\ \textsc{Axiom}$$

is equivalent to

$$\frac{\dfrac{}{\phi^L, \phi^R}\ \textsc{Hyp} \qquad \dfrac{}{\psi^L, \psi^R}\ \textsc{Hyp}}{\phi^L, \psi^R}\ \textsc{AxiomCut} \quad (\text{with } (\phi, \psi) \in A).$$

We now show that the Cut rule is admissible in $\mathbf{CF}^+$, that is, if $\mathcal{A}$ and $\mathcal{B}$ are proofs in $\mathbf{CF}^+$ of the sequents Γ, ψ^R and ψ^L, Δ, then there exists a proof in $\mathbf{CF}^+$ of the sequent Γ, Δ. The proof proceeds by induction, first on the size of ψ and then on the sum of the size of $\mathcal{A}$ and $\mathcal{B}$. Hence, we can assume that the Cut rule is admissible for proofs $\mathcal{A}'$, $\mathcal{B}'$, where $\mathcal{A}'$ is a proof of Γ', ψ'^R and $\mathcal{B}'$ is a proof of ψ'^L, Δ', and either ψ' is smaller than ψ or the sum of the sizes of $\mathcal{A}'$ and $\mathcal{B}'$ is smaller than the sum of the sizes of $\mathcal{A}$ and $\mathcal{B}$.

Case 1: Hyp Suppose $\mathcal{A}$ is a single Hyp step. Then, $\Gamma = \psi^L$ and hence $\mathcal{B}$ is a proof of Γ, Δ. The case where $\mathcal{B}$ is an instance of Hyp is symmetric.

Case 2: AxiomCut Suppose $\mathcal{A}$ ends with an AxiomCut rule.

$$\frac{\dfrac{\dfrac{\mathcal{A}'}{\Gamma, s} \quad \dfrac{\mathcal{A}''}{t, \psi^R}\ \textsc{AxiomCut}\ (s,t)}{\Gamma, \psi^R} \quad \dfrac{\mathcal{B}}{\psi^L, \Delta}\ \textsc{Cut}}{\Gamma, \Delta} \quad \hookrightarrow \quad \frac{\dfrac{\mathcal{A}'}{\Gamma, s} \quad \dfrac{\dfrac{\mathcal{A}''}{t, \psi^R} \quad \dfrac{\mathcal{B}}{\psi^L, \Delta}\ \textsc{Cut}}{t, \Delta}\ \textsc{AxiomCut}\ (s,t)}{\Gamma, \Delta}$$

The case where $\mathcal{B}$ ends with an AxiomCut rule is symmetric.

Case 3: left or right rule with non-principal formula In this case, we consider all situations where $\mathcal{A}$ ends with a left or right rule, and ψ^R is not the principal formula.

Case 3.a: LeftAnd Suppose $\mathcal{A}$ ends with a LeftAnd and $\Gamma = (\alpha \wedge \beta)^L$.

$$\frac{\dfrac{\dfrac{\mathcal{A}'}{\alpha^L, \psi^R}}{(\alpha \wedge \beta)^L, \psi^R}\ \textsc{LeftAnd} \quad \dfrac{\mathcal{B}}{\psi^L, \Delta}\ \textsc{Cut}}{(\alpha \wedge \beta)^L, \Delta} \quad \hookrightarrow \quad \frac{\dfrac{\dfrac{\mathcal{A}'}{\alpha^L, \psi^R} \quad \dfrac{\mathcal{B}}{\psi^L, \Delta}\ \textsc{Cut}}{\alpha^L, \Delta}\ \textsc{LeftAnd}}{(\alpha \wedge \beta)^L, \Delta}$$

Case 3.b: LeftOr Suppose $\mathcal{A}$ ends with a LeftOr and $\Gamma = (\alpha \vee \beta)^L$.

$$\dfrac{\dfrac{\dfrac{\mathcal{A}'}{\alpha^L, \psi^R} \quad \dfrac{\mathcal{A}''}{\beta^L, \psi^R}}{(\alpha \vee \beta)^L, \psi^R}\,\textsc{LeftOr} \quad \dfrac{\mathcal{B}}{\psi^L, \Delta}}{(\alpha \vee \beta)^L, \Delta}\,\textsc{Cut}$$

$$\hookrightarrow$$

$$\dfrac{\dfrac{\mathcal{A}'}{\alpha^L, \psi^R} \quad \dfrac{\mathcal{B}}{\psi^L, \Delta}}{\alpha^L, \Delta}\,\textsc{Cut} \quad \dfrac{\dfrac{\mathcal{A}''}{\beta^L, \psi^R} \quad \dfrac{\mathcal{B}}{\psi^L, \Delta}}{\beta^L, \Delta}\,\textsc{Cut}}{(\alpha \vee \beta)^L, \Delta}\,\textsc{LeftOr}$$

Case 3.c: LeftNot Suppose $\mathcal{A}$ ends with a LeftNot rule, i.e. $\Gamma = (\neg\alpha)^L$.

$$\dfrac{\dfrac{\dfrac{\mathcal{A}'}{\alpha^R, \psi^R}}{(\neg\alpha)^L, \psi^R}\,\textsc{LeftNot} \quad \dfrac{\mathcal{B}}{\psi^L, \Delta}}{(\neg\alpha)^L, \Delta}\,\textsc{Cut} \quad \hookrightarrow \quad \dfrac{\dfrac{\dfrac{\mathcal{A}'}{\alpha^R, \psi^R} \quad \dfrac{\mathcal{B}}{\psi^L, \Delta}}{\alpha^R, \Delta}\,\textsc{Cut}}{(\neg\alpha)^L, \Delta}\,\textsc{LeftNot}$$

The cases where $\mathcal{A}$ ends with a RightAnd, RightOr or RightNot rule are symmetric to the cases above, and can be transformed in the same way. Similarly, if $\mathcal{B}$ ends with a LeftAnd, LeftOr, LeftNot, RightAnd, RightOr or RightNot rule and ψ^L is not the principal formula, the transformation is symmetric to the cases above.

Case 4: right rule with principal formula Suppose $\mathcal{A}$ ends with a right rule and ψ^R is principal. We have already treated the case where $\mathcal{B}$ ends with an AxiomCut or Hyp rule or a left or right rule where ψ is not principal. Hence, we are left to consider the cases where $\mathcal{B}$ ends with a left rule or Replace rule. We assume here that $\mathcal{B}$ does not end with a Replace rule, as this case is treated in case 7, and hence we have three cases to consider.

Case 4.a If $\mathcal{A}$ ends with a RightOr rule and $\mathcal{B}$ ends with a LeftOr rule, i.e. $\psi = (\alpha \vee \beta)^R$.

$$\dfrac{\dfrac{\dfrac{\mathcal{A}'}{\Gamma, \alpha^R}}{\Gamma, (\alpha \vee \beta)^R}\,\textsc{RightOr} \quad \dfrac{\dfrac{\mathcal{B}'}{\alpha^L, \Delta} \quad \dfrac{\mathcal{B}''}{\beta^L, \Delta}}{(\alpha \vee \beta)^L, \Delta}\,\textsc{LeftOr}}{\Gamma, \Delta}\,\textsc{Cut} \quad \hookrightarrow \quad \dfrac{\dfrac{\mathcal{A}'}{\Gamma, \alpha^R} \quad \dfrac{\mathcal{B}'}{\alpha^L, \Delta}}{\Gamma, \Delta}\,\textsc{Cut}$$

Case 4.b If $\mathcal{A}$ ends with a RightAnd rule and $\mathcal{B}$ ends with a LeftAnd rule then the proof is dual to case 4.a.

Case 4.c If $\mathcal{A}$ ends with a RightNot rule and $\mathcal{B}$ ends with a LeftNot rule, i.e. $\psi = (\neg\alpha)^R$.

$$\dfrac{\dfrac{\dfrac{\mathcal{A}'}{\Gamma, \alpha^L}}{\Gamma, (\neg\alpha)^R}\,\textsc{RightNot} \quad \dfrac{\dfrac{\mathcal{B}'}{\alpha^R, \Delta}}{(\neg\alpha)^L, \Delta}\,\textsc{LeftNot}}{\Gamma, \Delta}\,\textsc{Cut} \quad \hookrightarrow \quad \dfrac{\dfrac{\mathcal{B}'}{\Delta, \alpha^R} \quad \dfrac{\mathcal{A}'}{\alpha^L, \Gamma}}{\Gamma, \Delta}\,\textsc{Cut}$$

Case 5: Replace We consider the cases where $\mathcal{A}$ ends with a Replace rule.

Case 5.a ψ^R is the principal formula.

$$\dfrac{\dfrac{\dfrac{\mathcal{A}'}{\Gamma, \Gamma}}{\Gamma, \psi^R}\,\textsc{Replace} \quad \dfrac{\mathcal{B}}{\psi^L, \Delta}}{\Gamma, \Delta}\,\textsc{Cut} \quad \hookrightarrow \quad \dfrac{\dfrac{\mathcal{A}'}{\Gamma, \Gamma}}{\Gamma, \Delta}\,\textsc{Replace}$$

If ψ^R is not the principal formula, then $\mathcal{A}$ has the shape:

$$\dfrac{\dfrac{\vdots}{\psi^R, \psi^R}}{\psi^R, \Delta}\,\textsc{Replace}$$

The rules that can conclude with ψ^R, ψ^R are only AxiomCut or a right rule.

Case 5.b ψ^R is not the principal formula and $\mathcal{A}$ ends with an AxiomCut rule and then Replace.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{A}'}{\psi^R, s} \quad \dfrac{\mathcal{A}''}{t, \psi^R}}{\psi^R, \psi^R}\,\textsc{AxiomCut}\ (s,t)}{\Gamma, \psi^R}\,\textsc{Replace} \quad \dfrac{\mathcal{B}}{\psi^L, \Delta}}{\Gamma, \Delta}\,\textsc{Cut}$$

$$\hookrightarrow$$

$$\dfrac{\dfrac{\dfrac{\mathcal{B}}{\Delta, \psi^L} \quad \dfrac{\mathcal{A}'}{\psi^R, s}}{\Delta, s}\,\textsc{Cut} \quad \dfrac{\dfrac{\mathcal{A}''}{t, \psi^R} \quad \dfrac{\mathcal{B}}{\psi^L, \Delta}}{t, \Delta}\,\textsc{Cut}}{\dfrac{\Delta, \Delta}{\Gamma, \Delta}\,\textsc{Replace}}\,\textsc{AxiomCut}\ (s,t)$$

Case 5.c $\mathcal{A}$ ends with Replace preceded by a right rule. We analyse $\mathcal{B}$. The only cases we have not yet considered are the cases where $\mathcal{B}$ ends with a left rule that has ψ^L as principal formula, or a Replace rule. We delay the case where $\mathcal{B}$ ends with a Replace rule to case 6.

Case 5.c.a $\mathcal{A}$ ends with RightOr then Replace, and $\mathcal{B}$ ends with LeftOr where $\psi = (\alpha \vee \beta)$ is principal.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{A}'}{(\alpha \vee \beta)^R, \alpha^R}}{(\alpha \vee \beta)^R, (\alpha \vee \beta)^R}\,\textsc{RightOr}}{\Gamma, (\alpha \vee \beta)^R}\,\textsc{Replace} \qquad \dfrac{\dfrac{\mathcal{B}'}{\alpha^L, \Delta} \quad \dfrac{\mathcal{B}''}{\beta^L, \Delta}}{(\alpha \vee \beta)^L, \Delta}\,\textsc{LeftOr}}{\Gamma, \Delta}\,\textsc{Cut}$$

$$\hookrightarrow$$

$$\dfrac{\dfrac{\dfrac{\mathcal{A}'}{(\alpha \vee \beta)^R, \alpha^R} \quad \dfrac{\mathcal{B}}{(\alpha \vee \beta)^L, \Delta}}{\Delta, \alpha^R}\,\textsc{Cut} \qquad \dfrac{\mathcal{B}'}{\alpha^L, \Delta}}{\dfrac{\Delta, \Delta}{\Gamma, \Delta}\,\textsc{Replace}}\,\textsc{Cut}$$

The first instance of Cut is justified by the fact that $|\mathcal{A}'| + |\mathcal{B}| \leq |\mathcal{A}| + |\mathcal{B}|$, and the second by the fact that the principal formula α is smaller than ψ.

Case 5.c.b $\mathcal{A}$ ends with RightAnd then Replace, and $\mathcal{B}$ ends with LeftAnd where $\psi = (\alpha \wedge \beta)$ is principal.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{A}'}{(\alpha \wedge \beta)^R, \alpha^R} \quad \dfrac{\mathcal{A}''}{(\alpha \wedge \beta)^R, \beta^R}}{(\alpha \wedge \beta)^R, (\alpha \wedge \beta)^R}\,\textsc{RightAnd}}{\Gamma, (\alpha \wedge \beta)^R}\,\textsc{Replace} \qquad \dfrac{\dfrac{\mathcal{B}'}{\alpha^L, \Delta}}{(\alpha \wedge \beta)^L, \Delta}\,\textsc{LeftAnd}}{\Gamma, \Delta}\,\textsc{Cut}$$

$$\hookrightarrow$$

$$\dfrac{\dfrac{\dfrac{\mathcal{A}'}{(\alpha \wedge \beta)^R, \alpha^R} \quad \dfrac{\mathcal{B}}{(\alpha \wedge \beta)^L, \Delta}}{\Delta, \alpha^R}\,\textsc{Cut} \qquad \dfrac{\mathcal{B}'}{\alpha^L, \Delta}}{\dfrac{\Delta, \Delta}{\Gamma, \Delta}\,\textsc{Replace}}\,\textsc{Cut}$$

As in the previous case, the first instance of Cut is justified by the fact that $|\mathcal{A}'| + |\mathcal{B}| \leq |\mathcal{A}| + |\mathcal{B}|$, and the second by the fact that the principal formula α is smaller than ψ.

Case 5.c.c $\mathcal{A}$ ends with RightNot then Replace, and $\mathcal{B}$ ends with LeftNot where $\psi = (\neg\alpha)$ is principal.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{A}'}{(\neg\alpha)^R, \alpha^L}}{(\neg\alpha)^R, (\neg\alpha)^R}\,\textsc{RightNot}}{\Gamma, (\neg\alpha)^R}\,\textsc{Replace} \qquad \dfrac{\dfrac{\mathcal{B}'}{\alpha^R, \Delta}}{(\neg\alpha)^L, \Delta}\,\textsc{LeftNot}}{\Gamma, \Delta}\,\textsc{Cut}$$

$$\hookrightarrow$$

$$\dfrac{\dfrac{\mathcal{B}'}{\alpha^R, \Delta} \qquad \dfrac{\dfrac{\mathcal{A}'}{(\neg\alpha)^R, \alpha^L} \qquad \dfrac{\mathcal{B}}{(\neg\alpha)^L, \Delta}}{\Delta, \alpha^L}\,\textsc{Cut}}{\dfrac{\Delta, \Delta}{\Gamma, \Delta}\,\textsc{Replace}}\,\textsc{Cut}$$

Once again, the first instance of Cut is justified by the fact that $|\mathcal{A}'|+|\mathcal{B}| \leq |\mathcal{A}|+|\mathcal{B}|$, and the second by the fact that the principal formula α is smaller than ψ.

Case 6 Both $\mathcal{A}$ and $\mathcal{B}$ end with a Replace rule, and ψ is the principal formula in both. We only have left to consider the cases where $\mathcal{A}$ ends with a right rule then Replace and $\mathcal{B}$ ends with a left rule then Replace.

Case 6.a If $\mathcal{A}$ ends with RightOr then Replace, then $\mathcal{B}$ must end with LeftOr then Replace and $\psi = (\alpha \vee \beta)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{A}'}{(\alpha\vee\beta)^R, \alpha^R}}{(\alpha\vee\beta)^R, (\alpha\vee\beta)^R}\,\textsc{RightOr}}{\Gamma, (\alpha\vee\beta)^R}\,\textsc{Replace} \qquad \dfrac{\dfrac{\dfrac{\mathcal{B}'}{\alpha^L, (\alpha\vee\beta)^L} \qquad \dfrac{\mathcal{B}''}{\beta^L, (\alpha\vee\beta)^L}}{(\alpha\vee\beta)^L, (\alpha\vee\beta)^L}\,\textsc{LeftOr}}{(\alpha\vee\beta)^L, \Delta}\,\textsc{Replace}}{\Gamma, \Delta}\,\textsc{Cut}$$

$$\hookrightarrow$$

$$\dfrac{\dfrac{\dfrac{\mathcal{A}'}{(\alpha\vee\beta)^R, \alpha^R} \qquad \dfrac{\mathcal{B}}{(\alpha\vee\beta)^L, \Delta}}{\Delta, \alpha^R}\,\textsc{Cut} \qquad \dfrac{\dfrac{\mathcal{A}}{\Gamma, (\alpha\vee\beta)^R} \qquad \dfrac{\mathcal{B}'}{\alpha^L, (\alpha\vee\beta)^L}}{\alpha^L, \Gamma}\,\textsc{Cut}}{\Gamma, \Delta}\,\textsc{Cut}$$

Here, the two top cuts are justified by the fact that $|\mathcal{A}'|+|\mathcal{B}| < |\mathcal{A}|+|\mathcal{B}|$ and $|\mathcal{A}|+|\mathcal{B}'| < |\mathcal{A}| + |\mathcal{B}|$. For the third cut, the cut formula α is smaller than ψ.

Case 6.b If $\mathcal{A}$ ends with RightAnd then Replace, then $\mathcal{B}$ must end with LeftAnd then Replace and $\psi = (\alpha \wedge \beta)^L$. This case is symmetric to the previous one, and can be transformed in the same way.

Case 6.c If $\mathcal{A}$ ends with RightNot then Replace, then $\mathcal{B}$ must end with LeftNot then Replace and $\psi = (\neg\alpha)^R$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{A}'}{(\neg\alpha)^R, \alpha^L}\,\textsc{RightNot}}{(\neg\alpha)^R, (\neg\alpha)^R}\,\textsc{Replace}}{\Gamma, (\neg\alpha)^R} \qquad \dfrac{\dfrac{\dfrac{\mathcal{B}'}{\alpha^L, (\neg\alpha)^L}\,\textsc{LeftNot}}{(\neg\alpha)^L, (\neg\alpha)^L}\,\textsc{Replace}}{(\neg\alpha)^L, \Delta}}{\Gamma, \Delta}\,\textsc{Cut}$$

$$\hookrightarrow$$

$$\dfrac{\dfrac{\dfrac{\mathcal{B}}{(\neg\alpha)^L, \Delta} \qquad \dfrac{\mathcal{A}'}{(\neg\alpha)^R, \alpha^L}\,\textsc{Cut}}{\Delta, \alpha^L} \qquad \dfrac{\dfrac{\mathcal{A}}{\Gamma, (\neg\alpha)^R} \qquad \dfrac{\mathcal{B}'}{\alpha^R, (\neg\alpha)^L}\,\textsc{Cut}}{\alpha^R, \Gamma}}{\Gamma, \Delta}\,\textsc{Cut}$$

Once again, the two top cuts are justified by the fact that $|\mathcal{A}'| + |\mathcal{B}| < |\mathcal{A}| + |\mathcal{B}|$ and $|\mathcal{A}| + |\mathcal{B}'| < |\mathcal{A}| + |\mathcal{B}|$, while the third instance of Cut is justified by the fact that the cut formula α is smaller than ψ.

Case 7: F-rule Finally, consider the case where $\mathcal{A}$ ends with the F-rule, and $\psi = F(\beta_x, \beta_y, \beta_z)^R$. For simplicity, we assume that F has arity 3 and takes one invariant parameter x, one monotonic parameter y and one antimonotonic parameter z.

$$\dfrac{\dfrac{\alpha_x^L, \beta_x^R \qquad \beta_x^L, \alpha_x^R \qquad \alpha_y^L, \beta_y^R \qquad \beta_z^L, \alpha_z^R}{F(\alpha_x, \alpha_y, \alpha_z)^L, F(\beta_x, \beta_y, \beta_z)^R}\,\text{F-rule} \qquad \dfrac{\mathcal{B}}{F(\beta_x, \beta_y, \beta_z)^L, \Delta}}{F(\alpha_x, \alpha_y, \alpha_z)^L, \Delta}\,\textsc{Cut}$$

We abbreviate $F(\alpha_x, \alpha_y, \alpha_z)$ as $F(\vec{\alpha})$, and $F(\beta_x, \beta_y, \beta_z)$ as $F(\vec{\beta})$. Now, consider the shape of $\mathcal{B}$. We have already covered the cases where $\mathcal{B}$ ends with a Hyp, AxiomCut, a left or right rule where ψ is not the principal formula. Note also that ψ cannot be the principal formula of a left or right rule. We are hence left to consider the cases where $\mathcal{B}$ ends with a Replace rule or an F-rule.

Case 7.a $\mathcal{B}$ ends with a Replace rule. Note that we have already covered the case where ψ is the principal formula in case 5.a. Hence, we can assume that $\mathcal{B}$ has shape:

$$\dfrac{\dfrac{\vdots}{F(\vec{\beta})^L, F(\vec{\beta})^R}\,\textsc{Replace}}{F(\vec{\beta})^L, \Delta}$$

There is only one rule that can conclude with $F(\vec{\beta})^L, F(\vec{\beta})^R$, namely AxiomCut. Hence:

$$\dfrac{\dfrac{\mathcal{A}'}{F(\vec{\alpha})^L, F(\vec{\beta})^R} \qquad \dfrac{\dfrac{\dfrac{\mathcal{B}'}{F(\vec{\beta})^L, s} \quad \dfrac{\mathcal{B}''}{t, F(\vec{\beta})^L}}{F(\vec{\beta})^L, F(\vec{\beta})^L}\ \textsc{AxiomCut}\ (s,t)}{F(\vec{\beta})^L, \Delta}\ \textsc{Replace}}{F(\vec{\alpha})^L, \Delta}\ \textsc{Cut}$$

$$\hookrightarrow$$

$$\dfrac{\dfrac{\dfrac{\mathcal{A}'}{F(\vec{\alpha})^L, F(\vec{\beta})^R} \quad \dfrac{\mathcal{B}'}{F(\vec{\beta})^L, s}\ \textsc{Cut} \qquad \dfrac{\dfrac{\mathcal{B}''}{t, F(\vec{\beta})^L} \quad \dfrac{\mathcal{A}'}{F(\vec{\alpha})^L, F(\vec{\beta})^R}\ \textsc{Cut}}{t, F(\vec{\alpha})^L}\ \textsc{AxiomCut}\ (s,t)}{F(\vec{\alpha})^L, s}\ \textsc{Cut}}{F(\vec{\alpha})^L, F(\vec{\alpha})^L}\ \textsc{Replace}}{F(\vec{\alpha})^L, \Delta}\ \textsc{Replace}$$

Case 7.b $\mathcal{B}$ ends with an G-RULE for some function symbol G. However because ϕ must be both of the form $F(...)^L$ and $G(...)^R$, we must have $F = G$, and $\mathcal{B}$ ends with F-RULE. Hence, we have:

$$\dfrac{\dfrac{\dfrac{\mathcal{A}_1}{\alpha_x^L, \beta_x^R} \quad \dfrac{\mathcal{A}_2}{\beta_x^L, \alpha_x^R} \quad \dfrac{\mathcal{A}_3}{\alpha_y^L, \beta_y^R} \quad \dfrac{\mathcal{A}_4}{\beta_z^L, \alpha_z^R}\ \textsc{F-rule}}{F(\alpha_x, \alpha_y, \alpha_z)^L, F(\beta_x, \beta_y, \beta_z)^R} \qquad \dfrac{\dfrac{\mathcal{B}_1}{\beta_x^L, \theta_x^R} \quad \dfrac{\mathcal{B}_2}{\theta_x^L, \beta_x^R} \quad \dfrac{\mathcal{B}_3}{\beta_y^L, \theta_y^R} \quad \dfrac{\mathcal{B}_4}{\theta_z^L, \beta_z^R}\ \textsc{F-rule}}{F(\beta_x, \beta_y, \beta_z)^L, F(\theta_x, \theta_y, \theta_z)^R}}{F(\alpha_x, \alpha_y, \alpha_z)^L, F(\theta_x, \theta_y, \theta_z)^R}\ \textsc{Cut}$$

$$\hookrightarrow$$

$$\dfrac{\dfrac{\dfrac{\mathcal{A}_1}{\alpha_x^L, \beta_x^R} \quad \dfrac{\mathcal{B}_1}{\beta_x^L, \theta_x^R}\ \textsc{Cut} \qquad \dfrac{\mathcal{B}_2}{\theta_x^L, \beta_x^R} \quad \dfrac{\mathcal{A}_2}{\beta_x^L, \alpha_x^R}\ \textsc{Cut}}{\alpha_x^L, \theta_x^R} \qquad \dfrac{\dfrac{\mathcal{A}_3}{\alpha_y^L, \beta_y^R} \quad \dfrac{\mathcal{B}_3}{\beta_y^L, \theta_y^R}\ \textsc{Cut} \qquad \dfrac{\mathcal{B}_4}{\theta_z^L, \beta_z^R} \quad \dfrac{\mathcal{A}_4}{\beta_z^L, \alpha_z^R}\ \textsc{Cut}}{\theta_z^L, \alpha_z^R}}{F(\alpha_x, \alpha_y, \alpha_z)^L, F(\theta_x, \theta_y, \theta_z)^R}\ \textsc{F-rule}$$

□

Given the large number of possible combinations of shapes of $\mathcal{A}$ and $\mathcal{B}$, it may not be convincing that the proof truly considers them all. This is where mechanization shines, as the proof has been fully checked in the Rocq proof assistant (Section 2.10).

The above theorem gives us an inductive characterization of truth in orthologic: if a formula is a theorem of orthologic, it has to be so because of one of the rules of the proof system, CUT excluded. We will use this characterization many times: to show that orthologic admits the interpolation property in Section 2.5, to design a normal form algorithm for formulas in orthologic in Section 2.4, and crucially to construct an efficient decision procedure for the entailment problem in orthologic.

2.2.3 Polynomial-time decision procedure for $\mathbf{LO}^+$ proof search

Cut elimination (Theorem 2.2.10) and the characterization of orthologic sequents as containing at most two formulas are the two key technical ingredients allowing one to design an efficient decision procedure for the entailment problem in orthologic. First, Cut elimination implies a subformula property.

Theorem 2.2.11 (Subformula property for orthologic)**.** Let s be a sequent in orthologic. Then, if s has a proof in $\mathbf{LO}^+$ with axioms in A, then it has a proof where all intermediate sequents only contain formulas that are subformulas of the formulas in s and A.

Proof. Suppose s is provable in $\mathbf{LO}^+$. By Theorem 2.2.10, it has a proof in $\mathbf{CF}^+$. For rules Hyp, Replace, LeftBot, RightTop, LeftAnd, RightAnd, LeftOr, RightOr, LeftNot, RightNot and F-rule, that is all rules from $\mathbf{CF}^+$ except AxiomCut, all formulas in the premises are subformulas of the conclusion. For the AxiomCut rule, formulas in the two premises are either subformulas of the conclusion or axioms from A. Hence, in a proof of s without Cut, all intermediate sequents will only contain formulas that are subformulas of the formulas in s and A. □

This theorem along with the characterization of orthologic sequents as containing at most two formulas gives us a polynomial bound on the search space of orthologic proofs.

Theorem 2.2.12 (Orthologic proof search)**.** There is a proof search procedure for $\mathbf{LO}^+$ running in time $\mathcal{O}(n^2(1+|A|))$, where $n = \|A\| + \|s\|$ is the total size of the problem.

Proof. We proceed by reducing proof search to validity of a set of propositional Horn clauses. For a sequent (Γ, Δ), let $\mathcal{S}(\Gamma, \Delta)$ be the set of all sequents that can be built from subformulas of Γ, Δ and A. Note that there are at most $\|s\| + \|A\|$ such subformulas and twice as many annotated formulas, and hence

$$|\mathcal{S}(\Gamma, \Delta)| \leq 4(\|s\| + \|A\|)^2 = \mathcal{O}(n^2) \tag{2.1}$$

Now, consider $\mathcal{S}(\Gamma, \Delta)$ as a set of propositional variables representing whether the corresponding sequent has a proof. Then, observe that every instance of a deduction rule whose premises and conclusion are in $\mathcal{S}(\Gamma, \Delta)$ corresponds to a Horn clause with 0, 1 or 2 antecedents. Let clauses(Γ, Δ) be the set of all such clauses. It follows that a sequent (Γ, Δ) has a proof if and only if its corresponding variable is a logical consequence of all the clauses in clauses(Γ, Δ).

To bound the size of clauses(Γ, Δ), we now wish to count, for an arbitrary sequent $s \in \mathcal{S}(\Gamma, \Delta)$, how many Horn clauses can have s as their conclusion. We count, for a fixed $s = (\Gamma, \Delta)$, how many rule instances can conclude with s:

- One way using the Hyp rule, if $\Gamma = \Delta$
- Two ways using the Replace rule, from Δ, Δ and Γ, Γ

- Two ways at most for each of Γ and Δ using LeftOr or RightAnd (less for other left and right rules, and they are exclusive)

- $\|A\|$ ways using the AxiomCut rule, one for each axiom $a \in A$.

Hence, any sequent s can only be the conclusion of *at most* $7 + \|A\|$ rule instances, and it follows that $|\operatorname{clauses}(\Gamma, \Delta)| \leq |\mathcal{S}(\Gamma, \Delta)|(7 + |A|)$. Using $n = \|(\Gamma, \Delta)\| + \|A\|$ and Equation 2.1, we obtain

$$|\operatorname{clauses}(\Gamma, \Delta)| = \mathcal{O}(n^2(1 + |A|))$$

Note that each clause contains at most 3 literals (one conclusion and two antecedents).

By [29], entailment in Horn clauses can be decided in linear time using unit propagation. Hence, we conclude that deciding if an $\mathbf{LO}^+$-sequent has a proof is decidable in time $\mathcal{O}(n^2(1 + |A|))$.

Corollary 2.2.13. The entailment problem for orthologic with (anti)monotonic function symbols is decidable in time $\mathcal{O}(n^2(1 + |A|))$.

□

2.2.4 Sequent calculus for lattices

In the introduction, we claimed that sequent calculi can be defined for bounded and unbounded lattices by adopting similar syntactic restrictions. The cut elimination theorem (Theorem 2.2.10) similarly holds when restricting the language to bounded or unbounded lattices. These systems will be important in Subsection 2.4.1 to construct a normalization algorithm for lattices (which is an intermediate step towards normalization for ortholattices in Subsection 2.4.2).

Definition 2.2.14. The proof system $\mathbf{LO}^+_{\boldsymbol{BL}}$ for bounded lattices is defined as the sequent calculus where sequents contain exactly one left and right formula, and where the rules are as follows:

$$\frac{}{\phi^L, \phi^R}\ \textsc{Hyp}$$

$$\frac{\gamma^L, \psi^R \qquad \psi^L, \delta^R}{\gamma^L, \delta^R}\ \textsc{Cut}$$

$$\frac{}{\bot^L, \gamma^R}\ \textsc{LeftBot} \qquad \frac{}{\gamma^L, \top^R}\ \textsc{RightTop}$$

$$\frac{\phi^L, \gamma^R}{(\phi \wedge \psi)^L, \gamma^R}\ \textsc{LeftAnd} \qquad \frac{\gamma^L, \phi^R \qquad \gamma^L, \psi^R}{\gamma^L, (\phi \wedge \psi)^R}\ \textsc{RightAnd}$$

$$\frac{\phi^L, \gamma^R \qquad \psi^L, \gamma^R}{(\phi \vee \psi)^L, \gamma^R}\ \textsc{LeftOr} \qquad \frac{\gamma^L, \phi^R}{\gamma^L, (\phi \vee \psi)^R}\ \textsc{RightOr}$$

$$\frac{\phi^L_{x_1}, \psi^R_{x_1} \qquad \psi^L_{x_1}, \phi^R_{x_1} \quad ... \qquad \phi^L_{y_1}, \psi^R_{y_1} \quad ... \qquad \psi^L_{z_1}, \phi^R_{z_1} \quad ...}{F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)^L, F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)^R}\ \text{F-rule}$$

$$\frac{}{\gamma^L, \delta^R}\ \textsc{Axiom} \quad \text{with } (\gamma^L, \delta^R) \in A$$

The system $\mathbf{LO}^+_{\boldsymbol{L}}$ for unbounded lattices is defined as the sequent calculus obtained from $\mathbf{LO}^+_{\boldsymbol{BL}}$ by removing the LeftBot and RightTop rules. The two systems $\mathbf{CF}^+_{\boldsymbol{BL}}$ and $\mathbf{CF}^+_{\boldsymbol{L}}$ are defined analogously to $\mathbf{CF}^+$ by replacing from $\mathbf{LO}^+_{\boldsymbol{BL}}$ and $\mathbf{LO}^+_{\boldsymbol{L}}$ respectively the Axiom and Cut rules by the rule AxiomCut:

$$\frac{\gamma^L, \phi^R \qquad \psi^L, \delta^R}{\gamma^L, \delta^R}\ \textsc{AxiomCut} \quad (\text{with } (\phi, \psi) \in A)$$

Theorem 2.2.15 (Soundness and completeness of $\mathbf{LO}^+_{\boldsymbol{BL}}$ and $\mathbf{LO}^+_{\boldsymbol{L}}$)**.** The systems $\mathbf{LO}^+_{\boldsymbol{BL}}$ and $\mathbf{LO}^+_{\boldsymbol{L}}$ are sound and complete with respect to the class of bounded and unbounded lattices, respectively.

Proof. As in Theorem 2.2.6 and Theorem 2.2.5. □

The cut elimination theorem generalizes to the systems $\mathbf{LO}^+_{\boldsymbol{BL}}$ and $\mathbf{LO}^+_{\boldsymbol{L}}$ as well.

Lemma 2.2.16. A sequent has a proof in $\mathbf{LO}^+_{\boldsymbol{BL}}$ (respectively $\mathbf{LO}^+_{\boldsymbol{L}}$) if and only if it has a proof in $\mathbf{CF}^+_{\boldsymbol{BL}}$ (respectively $\mathbf{CF}^+_{\boldsymbol{L}}$).

Proof. As in Theorem 2.2.10. □

We briefly highlight the significance of the results above in terms of universal algebra.

Definition 2.2.17. Let $\dashv\vdash_{\mathbf{CF}^+}$ be the relation such that $S \dashv\vdash_{\mathbf{CF}^+} T$ if and only if both $\vdash_{\mathbf{CF}^+} S^L, T^R$ and $\vdash_{\mathbf{CF}^+} T^L, S^R$ hold. Define $\dashv\vdash_{\mathbf{CF}^+_{\boldsymbol{BL}}}$ and $\dashv\vdash_{\mathbf{CF}^+_{\boldsymbol{L}}}$ analogously.

Lemma 2.2.18 (Soundness and completeness of $\mathbf{CF}^+$, $\mathbf{CF}^+_{\boldsymbol{BL}}$ and $\mathbf{CF}^+_{\boldsymbol{L}}$)**.** For all $S, T \in \mathcal{T}_{OL^+}(X)$,

$$(S \sim_{OL^+} T) \iff (S \dashv\vdash_{\mathbf{CF}^+} T)$$

For all $S, T \in \mathcal{T}_{BL^+}(X)$,

$$(S \sim_{BL^+} T) \iff (S \dashv\vdash_{\mathbf{CF}^+_{BL}} T)$$

For all $S, T \in \mathcal{T}_{L^+}(X)$,

$$(S \sim_{L^+} T) \iff (S \dashv\vdash_{\mathbf{CF}^+_{L}} T)$$

Proof. For OL^+, follows from Theorem 2.2.5, Theorem 2.2.6 and Theorem 2.2.10.

For BL^+ and L^+, use Theorem 2.2.15 and Lemma 2.2.16. □

So $\mathcal{F}_{OL^+}(X)$ (the free algebra over OL^+) is isomorphic to $\mathcal{T}_{OL^+}(X)_{/\dashv\vdash_{\mathbf{CF}^+}}$. In essence, we have constructed a syntactic and efficiently computable representation of $\mathcal{F}_{OL^+}(X)$, where as a reminder OL^+ is the variety of ortholattices with (anti)monotonic function symbols satisfying an arbitrary finite set of axioms A. This again generalizes straightforwardly to L^+ and BL^+ (with axioms).

2.3 Entailment problem for orthologic in practice

In practice, the Horn-clause-based proof search procedure from Theorem 2.2.12 is rather inefficient. It has two phases: First, it constructs all Horn clauses of the problem, then it deduces true sequents until it proves the goal. So even when the sequent has a short proof that could be quickly found with a reasonable heuristic in phase two, the algorithm will always hit the theoretical worst case of $\mathcal{O}(n^2(1+|A|))$ operations in the first phase. For practical applications, it is highly desirable, if possible, to avoid constructing the full set of Horn clauses explicitly.

Additionally, in the particular case where there are no axioms, the algorithm can be significantly simplified into a backward proof search procedure. We present these alternative algorithms. In this section, we represent elements of $\mathcal{T}_{OL^+}(X)$ using the `Formula` data structure of Listing 2.1. We use `!phi`, `phi1 ∧ phi2` and `phi1 ∨ phi2` as shorthands for `Not(phi)`, `And(phi1, phi2)` and `Or(phi1, phi2)` respectively.

Acknowledgement of contributions The algorithm of Subsection 2.3.1 along with its complexity analysis is the result of joint work with Vladislas de Haldat du Lys.

2.3.1 Improved algorithm for orthologic entailment

Here we work with OL (without function symbols) rather than OL^+. First, it is convenient to compute the negation normal form (NNF) of formulas, restricting their syntax. For propositional formulas without function symbols, negation normal form is easy to compute in linear time. Note that in order for the algorithm to still run in linear time in the presence of structure sharing, we need to memoize the result of NNF for each subformula. Moreover, we will need later to compute the negation of a formula in negation normal form. Formally, we define

$$\begin{aligned}&\text{getInverse} : \mathcal{T}_{OL}(X) \to \mathcal{T}_{OL}(X)\\ &\text{getInverse}(\phi) := \text{NNF}(\neg\phi)\end{aligned}$$

We use ϕ' as a shorthand for $\text{getInverse}(\phi)$.

For any formula $\phi \in \mathcal{T}_{OL}(X)$, $\text{NNF}(\phi) \sim_{OL} \phi$, so to decide if a sequent $\phi^{\square}, \psi^{\triangle}$ is provable in OL, it is sufficient to decide if the sequent $\text{NNF}(\phi)^{\square}, \text{NNF}(\psi)^{\triangle}$ is provable. Hence, we can restrict our attention to formulas in negation normal form. Additionally, by soundness and completeness of the OL proof system (Theorem 2.2.5, Theorem 2.2.6), a sequent (ϕ^L, Δ) is provable if and only if $((\neg\phi)^R, \Delta)$ is provable. Hence, any sequent can be transformed, in linear time, into an equiprovable sequent with two right formulas so that negation only appears right on top of variables and function symbols.

Hence, by replacing the Hyp rule by the equivalent

$$\frac{}{x^R, \neg x^R}\ \textsc{Hyp}$$

and the Cut rule by

Listing 2.1: Data structure for formulas

```
class Formula:
  val uniqueId: Int = getFreshId() //formula counter

case class Var(id: String) extends Formula
case object Bot extends Formula
case object Top extends Formula
case class Not(ch: Formula) extends Formula
case class Or(ch: Formula, ch2: Formula) extends Formula
case class And(ch: Formula, ch2: Formula) extends Formula
case class Fun(name: String,
               xArgs: List[Formula],
               yArgs: List[Formula],
               zArgs: List[Formula]) extends Formula
```

$$\frac{\phi^R, \gamma^R \qquad \text{getInverse}(\gamma)^R, \psi^R}{\phi^R, \psi^R}\ \textsc{Cut}$$

we can ensure all sequents in a proof only ever have right formulas; in particular we can ignore the rules LeftOr, LeftAnd, LeftNot and RightNot. This means that our search space is not all pairs of left and right subformulas, but pairs of the subformulas of the NNF of the input and the NNF of their negations. From now on, we consider without loss of generality that sequents have only right formulas that are in NNF. Hence, the rules that we need to consider are Axiom, Hyp, Cut, Replace, $\wedge$-R, $\vee$-R. We denote by SF the set of subformulas of our problem and by AF the set of formulas that compose the axioms. By P, we denote the set of proven sequents, initially containing only the axioms and hypotheses.

The goal is to design an algorithm that can compute up to $\mathcal{O}(n^2)$ provable sequents in time at most $\mathcal{O}(n^2(1 + |A|))$ while limiting preprocessing as much as possible. To get an intuition for the problem, observe that if there are a constant number of axioms, then our asymptotic time allowance and number of deductions are equal; hence, we may generally only perform a constant number of operations for each deduced sequent, except when the Cut rule is involved.

Suppose we just proved the sequent (ϕ, ψ). We want to discover every sequent that follows from it and previously proven sequents by some rule. We explain informally how the adequate data structure for each rule is maintained before giving the full algorithm.

For Axiom and Hyp, we simply start the algorithm by adding to the proven list every axiom and every sequent of the form $(x, \neg x)$, for every variable x in the input.

For the Cut rule, that means finding all previously proven sequents that contain either $\text{getInverse}(\phi)$ or $\text{getInverse}(\psi)$. To do so, we maintain a map $P_{Cut} : (AF \cup AF') \to \mathcal{P}(SF)$ from axiom formulas to sets of formulas. P_{Cut} satisfies the following invariant:

$$P_{Cut}(a) = \{b \mid (\text{getInverse}(a), b) \in P\}$$

When proving (ϕ, γ), if $\gamma \in AF$, then it follows that for every formula $\psi \in P_{Cut}(\text{getInverse}(\gamma))$, (ϕ, ψ) is provable. Note that it may be that (ϕ, ψ) was already deduced from another cut formula, but there are at most $|A|$ such cut formulas.

Then, the RightAnd rule has two premises. First, we initialize a map

$$SF_\wedge : SF \to \mathcal{P}(SF)$$

$$SF_\wedge(a) = \{b \mid a \wedge b \in SF\}$$

We also maintain a different map

$$P_\wedge : SF \to SF \to \mathcal{P}(SF)$$

with the invariant $P_\wedge(b)(k) = \{a \wedge k \mid (a, b) \in P \text{ and } a \wedge k \in SF\}$.

When proving a sequent (a, b), for every $k \in SF_\wedge(a)$, we update

$$P_\wedge(b)(k) \texttt{ += } a \wedge k,$$

and symmetrically for b. Then, when deducing a sequent (ϕ, ψ), we can automatically deduce all sequents of the form (ϕ, γ) for $\gamma \in P_\wedge(\phi)(\psi)$.

The RightOr rule is even simpler, as it has only one premise. We initialize a map

$$SF_\vee : SF \to \mathcal{P}(SF)$$

$$SF_\vee(a) = \{b \mid a \vee b \in SF\}$$

When deducing a sequent (ϕ, ψ), then for all $\gamma \in SF_\vee(\psi)$ we should also deduce the sequent $(\phi, \psi \vee \gamma)$ (and similarly for ϕ).

For the Replace rule, which also has only one premise, when deducing a sequent (ϕ, ϕ), we can deduce the sequent (ψ, ϕ) for every $\psi \in SF$.

Listing 2.2 summarizes the algorithm. In it, every data structure we use is the mutable version and `X` is the set of variables. Moreover, pairs of formulas are, as in $\mathbf{LO}^+$, unordered pairs. The following theorem states its correctness.

Theorem 2.3.1 (Correctness of *OL* entailment algorithm)**.** The `prove` function of Listing 2.2 outputs `true` if and only if the sequent s has a proof from the axioms in A.

Proof. ($\Rightarrow$) Suppose the algorithm outputs `true` on the sequent s and the axioms A. Let us prove that the sequent s has a proof from axioms A. Before making any deductive step, the algorithm initializes the data structures. The sets *proven*, P_{Cut} and $P_\wedge$ are initialized as empty and the sets $SF_\wedge$ and $SF_\vee$ are computed once and are not meant to change afterwards. In the main loop, the algorithm takes a sequent (a, b) that has been proven but whose consequences have not yet been computed. Then, the map P_{Cut} is updated twice. Its invariant is as follows: For every subformula a, $P_{Cut}(a) = \{b \mid (\text{getInverse}(a), b) \in P\}$. Since $(a, b) \in P$, the invariant holds after each iteration. The situation is symmetric for $P_{Cut}(\text{getInverse}(b))$. The map $P_\wedge$ is also updated according to

Listing 2.2: Optimized Orthologic Entailment Algorithm

```
type Sequent = (Formula, Formula)

def prove(s: Sequent, axioms: Set[Sequent]): Boolean =
  val subformulas: Set[Formula] =
          subformulasOf(s) ++ axioms.flatMap(subformulasOf)
  val proven: Set[Sequent] = Set.empty
  val worklist: Stack[Sequent] =
          Stack.from(axioms ++ X.map(x ⇒ (x, getInverse(x))))
  val P_Cut: Map[Formula, Set[Formula]] = Map.empty
  val P_and: Map[Formula, Map[Formula, Set[Formula]]] = Map.empty

  val SF∨: Map[Formula, Set[Formula]] = Map.empty
  val SF∧: Map[Formula, Set[Formula]] = Map.empty
  subformulas.foreach :
    case φ: φ₁ ∧ φ₂ ⇒
      SF∧(φ₁) += φ₂
      SF∧(φ₂) += φ₁
    case φ: φ₁ ∨ φ₂ ⇒
      SF∨(φ₁) += φ₂
      SF∨(φ₂) += φ₁

  while worklist.nonEmpty do
    val (a, b) = worklist.pop()
    if (a, b) == s then return true
    P_Cut(getInverse(a)) += b
    P_Cut(getInverse(b)) += a

      SF∧(a).foreach{ ψ ⇒ P_and(b)(ψ) += a ∧ ψ}
      SF∧(b).foreach{ ψ ⇒ P_and(a)(ψ) += b ∧ ψ}

      (SF∨(a) ++ P_and(b)(a) ++ P_Cut(a)).foreach: φ ⇒
        if !proven.contains((φ, b)) then
          proven.add((φ, b))
          worklist.push((φ, b))
      (SF∨(b) ++ P_and(a)(b) ++ P_Cut(b)).foreach: φ ⇒
        if !proven.contains((a, φ)) then
          proven.add((a, φ))
          worklist.push((a, φ))

      if a == b then
        subformulas.foreach: φ ⇒
          if !proven.contains((φ, a)) then
            proven.add((φ, a))
            worklist.push((φ, a))

  return false
```

the following invariant: for all subformulas b, ψ, $P_\wedge(b)(\psi) = \{a \wedge \psi \mid (a, b) \in P \text{ and } a \wedge \psi \in SF\}$. By construction, $\psi \in SF_\wedge(a)$ implies $a \wedge \psi \in SF$. Furthermore, since $(a, b) \in P$, the invariant holds. The reasoning is symmetric for $P_\wedge(b)$. Once the data structures are updated, we deduce new sequents. There are three cases (and their duals) to handle:

1. For sequents of the shape (ϕ, b), where ϕ belongs to the union of $P_{Cut}(a)$, $SF_\vee(a)$ and $P_\wedge(b)(a)$, we proceed set by set. If ϕ is in $SF_\vee(a)$, then we have $\phi = a \vee x$ for a given $x \in SF$ and $(a \vee x, b)$ is valid since it follows from the (RightOr) rule. Otherwise, if $\phi \in P_{Cut}(a)$, then, according to P_{Cut}'s invariant, $(\text{getInverse}(a), \phi) \in P$. Thus, also having (a, b) valid, it is correct, using Cut rule, to deduce (ϕ, b). Finally, if $\phi = x \wedge a$ is in $P_\wedge(b)(a)$, then, according to $P_\wedge$'s invariant, $(x, b) \in P$ and $\phi \in SF$. Therefore, since (a, b) and (x, b) are valid, it is correct, using (RightAnd) rule, to deduce $(x \wedge a, b)$.

2. For sequents of the shape (a, ψ) the reasoning is symmetric.

3. In the last case, $a = b$ so using Replace every sequent of the form (a, ϕ) is valid.

Since the axiom and hypothesis sequents are valid sequents, the initialization of the worklist and P is correct as well and hence, by induction, all sequents in P are provable. ($\Leftarrow$) Conversely, suppose that the sequent s has a proof from the axioms A. We proceed by induction on the proof. To show that provable sequents are eventually added to P, if the proof is an instance of Axiom or of Hyp, s is in P by initialization, thus the algorithm yields `true`. Then:

1. (Cut rule) Suppose $s = (a, b)$ and the premises $s_1 = (a, c)$ and $s_2 = (\text{getInverse}(c), b)$. By induction, at some point $s_1, s_2 \in P$. If s_1 has been added to P before s_2, we have, according to line 27, that $a \in P_{Cut}(\text{getInverse}(c))$. Then when adding s_2 to P, we deduce, for all $\phi \in P_{Cut}(\text{getInverse}(c))$, the sequent (ϕ, b). Hence s gets added to the worklist. Hence, $s \in P$. The reasoning supposing s_2 is added to P before s_1 is symmetric.

2. (RightAnd) Suppose $s = (a \wedge k, b)$ and the premises $s_1 = (a, b)$ and $s_2 = (k, b)$. By induction, at some point $s_1, s_2 \in P$. If s_1 has been added to P before s_2, we have that $a \wedge k \in P_\wedge(b)(k)$, since $SF_\wedge(a) \subseteq SF$ and s is the goal sequent. Then, the treatment of $\mathsf{add}(s_2)$ deduces, for all $\phi \in P_\wedge(b)(k)$, the sequent (ϕ, b). Since $a \wedge k \in P_\wedge(b)(k)$, s gets added to the worklist (and later in P)

3. (RightOr) Suppose $s = (a, b \vee k)$ and the premise is $s_1 = (a, b)$. By induction, at some point $s_1 \in P$ and then the algorithm has deduced, for all $\psi \in SF_\vee(b)$, the sequent (a, ψ). Since $k \in SF_\vee(b)$, s is added to the worklist. The reasoning when $s_1 = (a, k)$ is symmetric.

4. (Replace rule) Suppose $s = (a, b)$ and the hypothesis $s_1 = (a, a)$. By induction hypothesis, $s_1 \in P$. Thus, after the block lines 43-47, the algorithm has deduced, for all $\phi \in SF$, the sequent (a, ϕ), and in particular s.

□

Theorem 2.3.2 (Complexity of entailment algorithm)**.** The algorithm in Listing 2.2 has a time complexity in $\mathcal{O}(n^2(1+|A|))$.

Proof. We divide the complexity analysis into two parts: the initialization, and the main loop. Clearly, the initialization of every data structure takes time either constant or $\mathcal{O}(|SF|)$, i.e., linear in the size of the input formulas.

For the main loop, first observe that its body will be executed at most once for every sequent $s = (a, b)$ where $a, b \in SF$. We analyse the time taken by the body of the loop across all sequents $s = (a, b)$, and split it into two parts: updating the data structure (lines 24 to 32) and adding new sequents to the worklist (lines 34 to 47).

The first block, between line 24 and 27, takes average constant time for every (a, b). Then, the `foreach` at line 30 loops over up to $|SF|$ elements, but observe that (since each conjunction contributes to exactly two sets)

$$\sum_{b \in SF} |SF_\wedge(b)| = \mathcal{O}(|SF|)$$

Hence, for a fixed a, the cost of this loop across all calls `add(a, b)` is $\mathcal{O}(|SF|)$ and hence the total cost across all possible sequents is $\mathcal{O}(|SF|^2)$. The loop line 31 is similar.

Then, lines 34 to 41 we deduce new sequents. The key observation here is that for every particular way of deducing a sequent s (that is, particular deduction rule and premises), there is exactly one time we will check if s has already been proven, and add it to the worklist if not. Indeed, let us count for every sequent s the number of times that the condition `if !proven.contains(s)` is checked. Note that here we are not summing over all executions of the body with unique (a, b), but combining all executions and summing over the s that we possibly try to deduce, that is:

1. (line 34, Cut) A sequent (ϕ, b) is deduced 1 time for every a such that (a, b) is deduced and $\phi \in P_{Cut}(a)$, so at most $|A|$ times since $P_{Cut}(a)$ is non-empty only when a is an axiom formula.

2. (line 34, RightAnd) A sequent $(\phi_1 \wedge \phi_2, b)$ is deduced 1 time only from either (ϕ_1, b) or (ϕ_2, b) (the second that is proven), since $\phi \in P_\wedge(b)(a)$ only if $a = \phi_1$ or $a = \phi_2$.

3. (line 34, RightOr) A sequent $(\phi_1 \vee \phi_2, b)$ is deduced 1 time at most from each of (ϕ_1, b) or (ϕ_2, b) since $\phi \in SF_\vee(a)$ only if $a = \phi_1$ or $a = \phi_2$.

4. (line 38) Again at most the same number of symmetric ways.

5. (line 43, Replace) A sequent (ϕ, ψ) is deduced 1 time from each of (ϕ, ϕ) and (ψ, ψ), so 2 times in total

Hence every sequent (ϕ, ψ) can be deduced in $\mathcal{O}(1+|A|)$ ways, hence the total number of formulas `phi` across all executions of the body considered in lines 37 and 41 is $\mathcal{O}(|SF|^2(1+$

$|A|)$), and so the total cost of lines 37-44 across all executions of the body is $\mathcal{O}(|SF|^2(1+|A|))$.

Note that if we had simply computed the worst-case complexity over a single execution of the loop and multiplied by n^2, we would have obtained a worse bound. For example, for a sequent (a, a) taken from the worklist, the execution of the loop takes linear time in $|SF|$ to deduce all sequents of the form (a, ϕ), so we would only obtain a cubic bound. □

Note on complexity of memoization and equality testing The complexity proof of Listing 2.2 relies on the fact that operations on data structures such as hash maps and hash sets can be done in average constant time, but this is a simplification. In reality, adding or removing an element from a hash set or hash map requires checking equality with other elements to ensure that there are no hash collisions. However, checking whether two formulas given as syntax trees are equal takes linear time, and hence would add a linear factor to the complexity of the algorithm.

Luckily, this can be avoided by using reference equality instead of structural equality.

Recall that by the subformula property, the algorithm only ever sees the $\mathcal{O}(n)$ different subformulas of the input and their negation normal form. We can fix one particular object in memory for each of these formulas, and assign a different unique identifier to each. Such identifiers require only $\mathcal{O}(\log(n))$ bits to represent. Then, if two terms have the same identifier, they must be structurally equal. Additionally, we can (but do not need to) enforce that structurally equal formulas share the same object in memory by memoizing the construction of formulas, a strategy known as hash-consing.

Concretely, we must ensure that throughout the algorithm, we do not create new formula objects. In particular, computing `getInverse = nnf(Not(a))` creates a new node `Not(a)`, and then again $\mathcal{O}(|a|)$ new nodes when computing its negation normal form. To avoid this, define `getInverse` for formulas in negation normal form as in Listing 2.3.

This guarantees we never instantiate any node beyond the n subnodes of the original formula (in negation normal form) and their inverse for a total of $2n$ nodes.

2.3.2 Merging axioms for quadratic complexity

In the particular case where $A = \emptyset$, the algorithms of Theorem 2.2.12 and the previous section are quadratic, which is the best-known result for the word problem in both ortholattices and lattices (see [98]). In general, it can be beneficial to keep the number of axioms as small as possible. For this purpose, we can combine axioms with the same left-hand side into one. Given two axioms representing $a \leq b_1$ and $a \leq b_2$, we can merge them into an equivalent one $a \leq b_1 \wedge b_2$. Indeed, given an axiom sequent $\{a^L, (b_1 \wedge b_2)^R\}$ we can derive $\{a^L, b_1^R\}$ as follows:

Listing 2.3: Negation in Negation Normal Form

```
class Formula :
  var inverse: Option[Formula] = None
...

/* Assumes φ is in negation normal form */
def getInverse(φ: Formula): Formula = {
  φ.inverse match
    case Some(value) ⇒ value
    case None ⇒
      val second = φ match
        case x: Var           ⇒ Not(x)
        case Top              ⇒ Bot
        case Bot              ⇒ Top
        case Not(x: Var)      ⇒ x
        case And(ch, ch2)     ⇒ Or(getInverse(ch), getInverse(ch2))
        case Or(ch, ch2)      ⇒ And(getInverse(ch), getInverse(ch2))
      φ.inverse = Some(second)
      second.inverse = Some(φ)
      second
  }
```

$$\dfrac{\dfrac{}{a^L, (b_1 \wedge b_2)^R}\,\textsc{Axiom} \qquad \dfrac{\dfrac{}{b_1^L, b_1^R}\,\textsc{Hyp}}{(b_1 \wedge b_2)^L, b_1^R}\,\textsc{LeftAnd}}{a^L, b_1^R}\,\textsc{Cut}$$

Dually, we can merge axioms with the same right-hand side, $a_1 \leq b$ and $a_2 \leq b$ into $a_1 \vee a_2 \leq b$. Finally, $a \leq \neg b$ can be rewritten into $b \leq \neg a$ and vice versa. We can repeat this process until all left-hand sides and all right-hand sides of axioms are distinct, and no left side is a complement of a right side (we can even use normal forms for ortholattices to make such checks more general, as in Section 2.4). Such axiom preprocessing transformations do not change the set of provable formulas. They can be done in time $\mathcal{O}(n^2)$ and they reduce $|A|$ while not increasing n. Using such transformations can thus improve the cubic bound for certain kinds of axiom sets. As a special case, if all axioms have the form $\top \leq b_i$ and $a_i \leq \bot$ (corresponding to singleton sequents), we can combine them into a single axiom, obtaining $\mathcal{O}(n^2)$ complexity.

Corollary 2.3.3. If all axioms in A are singleton sequents, then the proof search in OL^+ has time complexity $\mathcal{O}(n^2)$.

2.3.3 Backward algorithms without axioms

In the absence of axioms, we can further simplify the data structures and algorithms for OL^+ and BL^+. In particular, if there are no axioms, we can fully eliminate the cut rule (Theorem 2.2.10). Then, the proof system $\mathbf{LO}^+_{\boldsymbol{BL}}$ (Definition 2.2.14) is algorithmic:

all rules can be applied from conclusion to premises, and the size of formulas always decreases, so that the process always terminates. This is essentially the generalization of Whitman's algorithm for lattices [98] to bounded lattices with function symbols. Concretely, Listing 2.4 decides the word problem for BL^+. Additionally, we found that it is typically more efficient in practice to use mutable fields for memoization rather than only external hash maps. We also found that using sparse bitsets provides a significant performance improvement over using hash sets. The data structure the algorithm uses for BL^+ is that of Listing 2.1 without the `Not` case, and with two fields for memoization:

```
class Formula:
  val uniqueId: Int = getFreshId() //formula counter
  val smaller = mutable.BitSet()    // contains φ s.t. this ⩽ φ
  val notSmaller = mutable.BitSet() // contains φ s.t. !(this ⩽ φ)
```

For OL^+, the situation is a little more complicated. First, as in Subsection 2.3.1, we use negation normal forms to restrict the shape of formulas and eliminate the need for LeftNot and RightNot rules. The main problem is that the system of Definition 2.2.3, even without Cut, Axiom, LeftNot, and RightNot, is not algorithmic. Indeed, the Replace rule does not guarantee that the size of formulas decreases and hence the proof search may not terminate. The following example illustrates an infinite branch of the proof search tree:

$$\dfrac{\dfrac{\dfrac{\vdots}{(a \vee b)^R, b^R}}{(a \vee b)^R, (a \vee b)^R}\,\textsc{RightOr}}{(a \vee b)^R, b^R}\,\textsc{Replace}$$

Observe, however, that when trying to prove a sequent Γ, Γ, it is not useful to try again an application of the Replace rule on Γ, as it would just lead back to the same sequent. We cannot simply add as an input to the recursive proof search function which sequents have already been tried, as it would lead to an exponential blow-up of the number of different inputs and break our memoization. Instead, we take two additional boolean parameters indicating, for each of the two formulas of the sequent, whether the Replace rule has already been tried on it. If it has, we skip the Replace rule for that formula. This ensures that between one use of the Replace rule and the next, the formulas have reduced in size and hence that the algorithm never loops. The cost to this is only to memoize four times as many entries, which is still asymptotically quadratic in the size of the input sequent. We implement and prove the soundness of this particular algorithm in Rocq in Section 2.10.

Listing 2.4: Computing $\leq$ for BL^+

```
def leq(φ: Formula, ψ: Formula): Boolean =
  // memoization hits
  if φ.smaller.contains(ψ.uniqueId) then return true
  if φ.notSmaller.contains(ψ.uniqueId) then return false

  // main recursive step
  val r: Boolean = (φ, ψ) match
    case (_, Top) ⇒ true
    case (Bot, _) ⇒ true
    case (Fun(name, xArgs, yArgs, zArgs),
          Fun(name2, xArgs2, yArgs2, zArgs2)) ⇒
      if name ≠ name2 then false
      else
        xArgs.zip(xArgs2).forall((x, x2) ⇒ leq(x, x2)) &&
        yArgs.zip(yArgs2).forall((y, y2) ⇒ leq(y2, y)) &&
        zArgs.zip(zArgs2).forall((z, z2) ⇒ leq(z, z2) && leq(z2, z))
    case (_, And(ψ₁, ψ₂)) ⇒
      leq(φ, ψ₁) && leq(φ, ψ₂)
    case (Or(φ₁, φ₂), _) ⇒
      leq(φ₁, ψ) && leq(φ₂, ψ)
    case (a, Or(ψ₁, ψ₂)) if isAtomOrFun(a) ⇒
      leq(a, ψ₁) || leq(a, ψ₂)
    case (And(φ₁, φ₂), b) if isAtomOrFun(b) ⇒
      leq(φ₁, b) || leq(φ₂, b)
    case (And(φ₁, φ₂), Or(ψ₁, ψ₂)) ⇒
      leq(φ₁, ψ) || leq(φ₂, ψ) ||
      leq(φ, ψ₁) || leq(φ, ψ₂)
    case (a, b) if isAtom(a) && isAtom(b) ⇒
      a == b
  // memoization store
  if r then φ.smaller.add(ψ.uniqueId)
  else φ.notSmaller.add(ψ.uniqueId)
  r

def isAtomOrFun(φ: Formula): Boolean = φ match
  case _: Var      ⇒ true
  case _: Fun      ⇒ true
  case _: Bot|Top  ⇒ true
  case _           ⇒ false
```

2.4 Normalization

In this section, we show that OL^+ admits an efficiently computable *normal form function*, which can be used to normalize types. The developments in this section generalize those of [32, 42, 14] to lattices and ortholattices with function symbols. Our proof makes heavy use of the proof theory developed in Section 2.2 to obtain an inductive principle to analyse truth in free lattices and ortholattices. This significantly simplifies the proofs and makes them self-contained, in contrast with the previous work cited above, which relied on abstract theorems from universal algebra.

Definition 2.4.1 (Minimal form)**.** Let $\mathcal{A}$ be an algebra. A term $\psi \in \mathcal{T}_{\mathcal{A}}(X)$ is in *minimal form* if there exists no ψ' such that $\|\psi'\| < \|\psi\|$ with $\psi' \sim_{\mathcal{A}} \psi$.

Definition 2.4.2 (Normal form function)**.** Let $\mathcal{A}$ be an algebra. For $\phi, \psi \in \mathcal{T}_{\mathcal{A}}(X)$, note that we use $=$ to denote the syntactic equality on trees. A *normal form function* is a function $N : \mathcal{T}_{\mathcal{A}}(X) \to \mathcal{T}_{\mathcal{A}}(X)$ satisfying the following three properties:

$$\begin{aligned}
&\forall \phi,\ N(\phi) \sim_{\mathcal{A}} \phi \\
&\forall \phi, \psi,\ \phi \sim_{\mathcal{A}} \psi \implies N(\phi) = N(\psi) \\
&\forall \phi,\ N(\phi) \text{ is in minimal form (Definition 2.4.1)}
\end{aligned}$$

2.4.1 Normalization for bounded lattices with functions

Our goal is to design a normal form function for OL^+, allowing for normalization of Boolean formulas with respect to ortholattice laws. The first step toward this is to construct a normal form function for BL^+, using the characterization of truth in BL^+ given by Lemma 2.2.18. We first state a useful inversion lemma for terms starting with a function symbol.

Lemma 2.4.3. Let $\psi \in \mathcal{T}_{BL^+}(X)$ be in minimal form and suppose

$$\psi \sim_{BL^+} F(\phi_{x_1}, \dots, \phi_{y_1}, \dots, \phi_{z_1}, ...)$$

where, as usual, $F(x_1, \dots, y_1, \dots, z_1, ...)$ is invariant in x_i, covariant in y_j and contravariant in z_k. Then $\psi = F(\psi_{x_1}, \dots, \psi_{y_1}, \dots, \psi_{z_1}, ...)$ and $\phi_{x_i} \sim_{BL^+} \psi_{x_i}$, $\phi_{y_j} \sim_{BL^+} \psi_{y_j}$, $\phi_{z_k} \sim_{BL^+} \psi_{z_k}$.

Proof. Consider the shape of ψ.

- For $\psi = x$, no rule of $\mathbf{CF}^+_{\boldsymbol{BL}}$ can conclude $F(\phi_{x_1}, \dots, \phi_{y_1}, \dots, \phi_{z_1}, ...) \leq_{BL^+} x$.
- Similarly, $F(\phi_{x_1}, \dots, \phi_{y_1}, \dots, \phi_{z_1}, ...) \leq_{BL^+} \bot$ and $\top \leq_{BL^+} F(\phi_{x_1}, \dots, \phi_{y_1}, \dots, \phi_{z_1}, ...)$ cannot be deduced.

- For $\psi = \psi_1 \vee \psi_2$, the only rule that can conclude $F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...) \leq_{BL^+} \psi_1 \vee \psi_2$ is RightOr and hence one of the ψ_i is such that $F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...) \leq_{BL^+} \psi_i$. But from $\psi_1 \vee \psi_2 \leq_{BL^+} F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)$, we deduce $\psi_i \leq_{BL^+} F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)$ and hence $\psi_i \sim_{BL^+} F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...) \sim_{BL^+} \psi$, contradicting the assumption that ψ is minimal.
- A dual argument similarly rules out $\psi = \psi_1 \wedge \psi_2$.
- For $\psi = g(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)$, note that the only rule that can deduce

  $$F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...) \leq_{BL^+} \psi$$

  if ψ has the shape of a function is the F-rule so $\psi = F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)$ and

  - $\phi_{x_i} \sim_{BL^+} \psi_{x_i}$
  - $\phi_{y_j} \leq_{BL^+} \psi_{y_j}$
  - $\psi_{z_k} \leq_{BL^+} \phi_{z_k}$.

  Symmetrically, $\psi \leq_{BL^+} F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)$ implies

  - $\psi_{y_j} \leq_{BL^+} \phi_{y_j}$
  - $\phi_{z_k} \leq_{BL^+} \psi_{z_k}$.

  So it follows that $\phi_{x_i} \sim_{BL^+} \psi_{x_i}$, $\phi_{y_j} \sim_{BL^+} \psi_{y_j}$ and $\phi_{z_k} \sim_{BL^+} \psi_{z_k}$.

□

Theorem 2.4.4. A formula $\psi \in \mathcal{T}_{BL^+}(X)$ is in normal form if and only if one of the following holds:

- $\psi \in X$
- $\psi \in \{\bot, \top\}$
- $\psi = F(\psi_1, \ldots, \psi_n)$ and each ψ_i is in normal form
- $\psi = \psi_1 \vee ... \vee \psi_n$, and all the following hold:
  - each ψ_i is in normal form
  - $\psi_i \not\leq_{BL^+} \psi_j$ for all $i \neq j$
  - if $\psi_i = \psi_{i1} \wedge ... \wedge \psi_{im}$ then for all j, $\psi_{ij} \not\leq_{BL^+} \psi$
- $\psi = \psi_1 \wedge ... \wedge \psi_n$ and the dual of the conditions above hold.

Proof. The forward direction is trivial. For the backward direction, let ϕ be in minimal form and such that $\phi \sim_{BL^+} \psi$. We show that $\phi = \psi$ by induction on the structure of ψ.

- If $\psi \in X \cup \{\bot, \top\}$ then trivially $\phi = \psi$.
- If $\psi = F(\psi_1, \ldots, \psi_n)$, by Lemma 2.4.3, $\phi = F(\phi_1, \ldots, \phi_n)$ and $\psi_i \sim_{BL^+} \phi_i$. Since by hypothesis ψ_i is in normal form and ϕ_i is in minimal form, by induction $\phi_i = \psi_i$ for all i and hence $\phi = \psi$.
- Suppose $\psi = \psi_1 \vee ... \vee \psi_n$.
  - If $\phi = x$ then $\psi_1 \vee ... \vee \psi_n \leq_{BL^+} x$ and hence necessarily $\forall i,\ \psi_i \leq_{BL^+} x$. Dually $x \leq_{BL^+} \psi_1 \vee ... \vee \psi_n$ and hence using $\mathbf{CF}^+_{\boldsymbol{BL}}$, there must be some ψ_i such that $x \leq_{BL^+} \psi_i$, so $\psi_i \sim_{BL^+} x \sim_{BL^+} \psi = \psi_1 \vee ... \vee \psi_n$, and in particular for any j, $\psi_j \leq_{BL^+} \psi_i$, contradicting the assumption.
  - If $\phi = \bot$ then for all i, $\psi_i \sim_{BL^+} \bot$, contradicting the assumption. If $\phi = \top$, then looking at $\mathbf{CF}^+_{\boldsymbol{BL}}$ there must be some i such that $\psi_i = \top$, again contradicting assumptions.
  - By Lemma 2.4.3, ϕ cannot start with a function symbol.
  - Now suppose that $\phi = \phi_1 \wedge ... \wedge \phi_m$. The only rules that can deduce $\phi \leq_{BL^+} \psi$ are LeftAnd and RightOr, and hence either

    $$(\exists i,\ \phi_i \leq_{BL^+} \psi) \text{ or } (\exists i,\ \phi \leq_{BL^+} \psi_i)$$

    In the first case, since $\phi_i \leq_{BL^+} \psi \sim_{BL^+} \phi$ and $\phi \leq_{BL^+} \phi_i$ we obtain $\phi_i \sim_{BL^+} \phi$, contradicting the minimality of ϕ. In the second case we have $\psi_1 \vee ... \vee \psi_n = \psi \sim_{BL^+} \phi \leq_{BL^+} \psi_i$, and in particular for all j, $\psi_j \leq_{BL^+} \psi_i$, contradicting the assumption.
  - Hence, we necessarily have $\phi = \phi_1 \vee ... \vee \phi_m \sim_{BL^+} \psi_1 \vee ... \vee \psi_n$, from which follows $\forall i,\ \phi_i \leq_{BL^+} \psi$.

    Now we want to show that $\forall i\ \exists j,\ \phi_i \leq \psi_j$. If $\phi_i \in X$ or ϕ_i starts with a function symbol, we must have $\exists j,\ \phi_i \leq \psi_j$ by inspecting the $\mathbf{CF}^+_{\boldsymbol{BL}}$ proof of $\phi_i \leq \psi_1 \vee ... \vee \psi_n$.

    On the other hand, if $\phi_i = \phi_{i1} \wedge ... \wedge \phi_{il}$ then we have

    $$\phi_{i1} \wedge ... \wedge \phi_{il} \leq \psi_1 \vee ... \vee \psi_n$$

    By inspecting $\mathbf{CF}^+_{\boldsymbol{BL}}$, we necessarily have either $\exists j,\ \phi_i \leq_{BL^+} \psi_j$ or $\exists j,\ \phi_{ij} \leq_{BL^+} \psi$, but the latter contradicts minimality of ϕ since we would have

    $$\phi \leq_{BL^+} \phi_1 \vee ... \vee \phi_{ij} \vee ... \vee \phi_m \leq_{BL^+} \psi \sim_{BL^+} \phi$$

    and in particular $\phi \sim_{BL^+} \phi_1 \vee ... \vee \phi_{ij} \vee ... \vee \phi_m$.

    By a similar argument, $\forall j\ \exists i,\ \phi_i \leq_{BL^+} \psi_j$ must hold as well. Since both are antichains, it must hold that $n = m$ and, after reordering, $\phi_i \sim_{BL^+} \psi_i$. Using induction, $\phi_i = \psi_i$, and hence $\phi = \psi$.

Listing 2.5: Data structure for formulas in BL^+.

```
class Formula:
  val uniqueId: Int = getFreshId() //formula counter
case class Var(id: String) extends Formula
case object Bot extends Formula
case object Top extends Formula
case class Or(children: List[Formula]) extends Formula
case class And(children: List[Formula]) extends Formula
case class Fun(name: String,
               xArgs: List[Formula],
               yArgs: List[Formula],
               zArgs: List[Formula]) extends Formula
```

- The $\psi = \psi_1 \wedge ... \wedge \psi_n$ case admits a dual proof.

□

Corollary 2.4.5. A formula $\phi \in \mathcal{T}_{BL^+}(X)$ is in normal form if and only if all the subterms ψ of ϕ of the form $\psi = \psi_1 \vee ... \vee \psi_n$ satisfy the two properties:

1. $\psi_i \not\leq_{BL^+} \psi_j$ for all $i \neq j$
2. if $\psi_i = \psi_{i1} \wedge ... \wedge \psi_{im}$ then for all j, $\psi_{ij} \not\leq_{BL^+} \psi$

and dually for subterms of the form $\psi = \psi_1 \wedge ... \wedge \psi_n$.

Theorem 2.4.6 (Normal form function for BL^+)**.** There exists an algorithm that takes as input $\phi \in \mathcal{T}_{BL^+}(X)$ a formula of size n, runs in time $\mathcal{O}(n^2)$ and outputs a normal form for ϕ.

Proof. The existence of a normalization algorithm for BL^+ follows from Corollary 2.4.5 and extends the normalization algorithm for lattices from [42].

First, assume that formulas are represented as syntax trees where conjunctions and disjunctions are n-ary nodes and flattened, so that for example the formula $(a \wedge b) \wedge (c \wedge d)$ is represented as a single $\wedge$ node with four children a, b, c, d. The corresponding data structure is given in Listing 2.5.

Then, define $\zeta : \mathcal{T}_{BL^+}(X) \to \mathcal{T}_{BL^+}(X)$ as given by Listing 2.6. In particular,

$$\zeta(a_1 \vee ... \vee a_m) = \begin{cases} \zeta(a_1 \vee ... \vee a_{ij} \vee ... \vee a_m) & \text{if } a_i = (a_{i1} \wedge ... \wedge a_{in}) \\ & \text{and } a_{ij} \leq_{BL^+} a_1 \vee ... \vee a_m \\ \zeta(a_1) \vee ... \vee \zeta(a_m) & \text{otherwise} \end{cases}$$

Note that ζ is idempotent and that $\forall \psi \in \mathcal{T}_{BL^+}(X),\ \psi \sim_{BL^+} \zeta(\psi)$. Let $\mathcal{T}_{BL^+}(X)^\zeta$ be the range of ζ. Note also that all terms in $\mathcal{T}_{BL^+}(X)^\zeta$ recursively satisfy the first property of Corollary 2.4.5.

Listing 2.6: Zeta normalization function

```
def zeta(φ: Formula): Formula = φ match
  case _: Var ⇒ φ
  case _: Bot|Top ⇒ φ
  case Or(disjuncts) ⇒
    var acc = List[Formula]()
    disjuncts.foreach {
      case φ_i @ And(conjuncts) ⇒
        // If there is φ_ij s.t. φ_ij ≤ φ, replace φ_i by φ_ij
        conjuncts.find(φ_ij ⇒ φ_ij ≤_BL+ φ) match
          case Some(φ_ij) ⇒ acc = zeta(φ_ij) :: acc
          case None       ⇒ acc = zeta(φ_i) :: acc
      case φ_i ⇒
        acc = zeta(φ_i) :: acc
    }
    Or(acc.reverse)

  case And(conjuncts) ⇒ ... // dual to the case above
  case Fun(f, xArgs, yArgs, zArgs) ⇒
    Fun(f, xArgs.map(zeta), yArgs.map(zeta), zArgs.map(zeta))
```

Now, define $\eta : \mathcal{T}_{BL^+}(X)^\zeta \to \mathcal{T}_{BL^+}(X)^\zeta$ according to Listing 2.7. In particular,

$$\eta(a_1 \vee ... \vee a_m) = \begin{cases} \eta(a_1 \vee ... \vee a_{i-1} \vee a_{i+1} \vee ... \vee a_m) & \text{if } a_i \leq_{BL^+} a_j,\ i \neq j \\ \eta(a_1) \vee ... \vee \eta(a_m) & \text{otherwise} \end{cases}$$

Observe again that η is idempotent and $\forall \psi \in \mathcal{T}_{BL^+}(X),\ \psi \sim_{BL^+} \eta(\psi)$, and let $\mathcal{T}_{BL^+}(X)^{\zeta\eta}$ be the range of η. Since η ensures that conjunctions and disjunctions are antichains, all elements of $\mathcal{T}_{BL^+}(X)^{\zeta\eta}$ satisfy the conditions of Corollary 2.4.5 and hence are in normal form.

Now let $\phi, \psi \in \mathcal{T}_{BL^+}(X)$ such that $\phi \sim_{BL^+} \psi$. Then

$$\eta(\zeta(\phi)) \sim_{BL^+} \phi \sim_{BL^+} \psi \sim_{BL^+} \eta(\zeta(\psi))$$

so $\eta(\zeta(\phi)) = \eta(\zeta(\psi))$.

Hence $\eta \circ \zeta$ is a normal form function. To analyse the runtime, observe that, if we ignore the time required to compute $\leq_{BL^+}$, η and ζ each take at most quadratic time to compute. Note also that $\leq_{BL^+}$ is only necessary to compute at most on every pair of subterms of the normal form of the input, so that the algorithm of Listing 2.4 (restricted to BL^+) computes all of them in time $\mathcal{O}(n^2)$. □

Listing 2.7: Eta normalization function

```
def eta(φ: Formula): Formula = φ match
  case _ : Var ⇒ φ
  case _ : Bot|Top ⇒ φ
  case Or(disjuncts) ⇒
    var antichain = List[Formula]()
    disjuncts.zipWithIndex.foreach { case (φ_i, i) ⇒
      if ! disjuncts.zipWithIndex.exists { case (φ_j, j) ⇒
          (φ_i ≤_BL+ φ_j) && (!(φ_j ≤_BL+ φ_i) || i > j)
        } then
        antichain = eta(φ_i) :: antichain
    }
    Or(antichain.reverse)
  case And(conjuncts) ⇒ ... // dual to the case above
  case Fun(f, xArgs, yArgs, zArgs) ⇒
    Fun(f, xArgs.map(eta), yArgs.map(eta), zArgs.map(eta))
```

2.4.2 Normalization for ortholattices with functions

To represent OL^+ formulas, we extend the data structure of Listing 2.5 with an additional case:

```
case class Not(child: Formula) extends Formula
```

The first technical difficulty for normalization in OL^+ relates to negation normal form. Indeed, for ortholattices without function symbols, NNF has the type:

$$\mathrm{NNF} : \mathcal{T}_{OL}(X) \to \mathcal{T}_{L}(X \cup X')$$

where $X' = \{\neg x \mid x \in X\}$, corresponding to negation normal form such that $[\mathrm{NNF}(\cdot)]_{OL} : \mathcal{T}_{OL^+}(X)_{/\sim_{OL}} \to \mathcal{T}_{L}(X \cup X')_{/\sim_{OL}}$ is an isomorphism of ortholattices. This is not immediately possible in OL^+, as negation cannot "go through" function symbols. Nonetheless, we can define $\mathrm{NNF} : \mathcal{T}_{OL^+}(X) \to \mathcal{T}_{OL^+}(X)$ as a pseudo negation-normal form, as in Listing 2.8.

Note that NNF is idempotent, and let $\mathcal{T}_{OL^+}(X)^{\mathrm{NNF}}$ be the range of NNF, i.e. the terms in pseudo negation-normal form. We now expand the signature of OL^+ to $\overline{OL^+}$.

Definition 2.4.7. Define $\overline{OL^+}$ (resp. $\overline{BL^+}$) as the algebraic signature of OL^+ (resp. BL^+) extended with, for each symbol F in OL^+, a new symbol $\bar{F}$ with the opposite monotonicity. Note that $\mathcal{T}_{OL^+}(X)^{\mathrm{NNF}}$ and $\mathcal{T}_{\overline{BL^+}}(X \cup X')$ are isomorphic, with the isomorphism induced by

$$\neg(F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, \ldots)) \mapsto \bar{F}(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, \ldots)$$

To reduce notational burden, we keep this isomorphism implicit and, with a slight abuse of notation, identify the expressions $\neg(F(x_1, \ldots, y_1, \ldots, z_1, \ldots))$ and

Listing 2.8: Negation Normal Form Computation

```
def nnf(φ): Formula = φ match
  case Var(x)         ⇒ Var(x)
  case Bot|Top     ⇒ φ
  case Not(ch)           ⇒ nnfNot(ch)
  case Or(disjuncts)     ⇒ Or(disjuncts.map(nnf))
  case And(conjuncts)     ⇒ And(conjuncts.map(nnf))
  case Fun(name, xArgs, yArgs, zArgs) ⇒
        Fun(name, xArgs.map(nnf), yArgs.map(nnf), zArgs.map(nnf))

def nnfNot(φ): Formula = φ match
  case Var(x)         ⇒ Not(Var(x))
  case Bot            ⇒ Top
  case Top            ⇒ Bot
  case Not(ch)            ⇒ nnf(ch)
  case Or(disjuncts)     ⇒ And(disjuncts.map(nnfNot))
  case And(conjuncts)     ⇒ Or(conjuncts.map(nnfNot))
  case Fun(name, xArgs, yArgs, zArgs) ⇒
        Not(Fun(name, xArgs.map(nnf), yArgs.map(nnf), zArgs.map(nnf)))
```

$\bar{F}(x_1, \ldots, y_1, \ldots, z_1, ...)$, in particular in code snippets.

Definition 2.4.8. We define a new proof system $\mathbf{CF}^+_{\delta}$ as the system made of all the rules of $\mathbf{CF}^+$ (Definition 2.2.9) except for the two negation rules, but with the Hyp rule extended to allow

$$\frac{}{x^L, \neg x^L} \quad \text{and} \quad \frac{}{x^R, \neg x^R}$$

for all $x \in X$.

Lemma 2.4.9. For all $\phi, \psi \in \mathcal{T}_{\overline{BL^+}}(X \cup X')$,

$$\vdash_{\mathbf{CF}^+} \phi^L, \psi^R \iff \vdash_{\mathbf{CF}^+_{\delta}} \phi^L, \psi^R$$

Proof. Forward: by induction on the proof of $\vdash_{\mathbf{CF}^+} \phi^L, \psi^R$. Note that all proof steps but LeftNot and RightNot over functions can be directly replicated in $\mathbf{CF}^+_{\delta}$. Then, since negation only occurs right above function symbols and variables, LeftNot and RightNot steps can only occur in the following configuration (the case where the negation appears above a variable is similar):

$$\dfrac{\dfrac{\dfrac{\mathcal{A}}{F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)^R, F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)^L}}{F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)^R, \neg F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)^R}\,\textsc{LeftNot}}{\neg F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)^L, \neg F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)^R}\,\textsc{RightNot}$$

Note that $\mathcal{A}$ is necessarily an instance of the F-rule, and hence the three steps can be replaced by one instance of the $\bar{F}$-rule.

Listing 2.9: Beta normalization function. `getInverse` comes from Listing 2.3

```
/* Assumes φ is in negation normal form */
def beta(φ: Formula): Formula = φ match
  case _: Variable ⇒ φ
  case _: Bot|Top ⇒ φ
  case Not(_: Variable) ⇒ φ
  case Or(disjuncts) ⇒
    if disjuncts.exists( φᵢ ⇒
      getInverse(φᵢ) ≤BL+ φ
    ) then Top
    else
      Or(disjuncts.map(beta))
  case And(conjuncts) ⇒
    ... // dual to the case above
  case Fun(f, xArgs, yArgs, zArgs) ⇒
    Fun(f, xArgs.map(beta), yArgs.map(beta), zArgs.map(beta))
```

Backward: by induction on the proof of $\vdash_{\mathbf{CF}^+_\delta} \phi^L, \psi^R$, the inverse transformation from the one above yields the desired proof of $\vdash_{\mathbf{CF}^+} \phi^L, \psi^R$. □

Lemma 2.4.9 allows us to obtain an nnf-like form where negations only appear above literals and function symbols. We now define one last reduction function. Let $\beta : \mathcal{T}_{\overline{BL^+}}(X \cup X') \to \mathcal{T}_{\overline{BL^+}}(X \cup X')$ according to Listing 2.9.

It is clear that $\forall \phi,\ \beta(\phi) \sim_{OL^+} \phi$, and hence in particular $\phi \leq_{OL^+} \psi \iff \beta(\phi) \leq_{OL^+} \beta(\psi)$. Let $\mathcal{T}_{\overline{BL^+}}(X \cup X')^\beta$ be the range of β.

Lemma 2.4.10. If $\phi \in \mathcal{T}_{\overline{BL^+}}(X \cup X')^\beta$, then

- $\phi \sim_{OL^+} \top \iff \phi \sim_{BL^+} \top$
- $\phi \sim_{OL^+} \bot \iff \phi \sim_{BL^+} \bot$

Proof. By induction on ϕ. Assuming $\phi \sim_{OL^+} \top$, ϕ cannot possibly be a variable or an applied function. If $\phi = \phi_1 \wedge \phi_2 \sim_{OL^+} \top$ then both $\phi_1 \sim_{OL^+} \top$ and $\phi_2 \sim_{OL^+} \top$, so by induction $\phi_1 \sim_{BL^+} \top$ and $\phi_2 \sim_{BL^+} \top$ and the conclusion follows. If $\phi = \phi_1 \vee \phi_2 \sim_{OL^+} \top$ then $\phi = \top$ by definition of β.

The case $\phi \sim_{OL^+} \bot$ is, as always, dual. □

We are now left with showing that $\mathcal{T}_{\overline{BL^+}}(X \cup X')^\beta$ is an OL^+ *when seen with the rules of* $\mathbf{CF}^+_{\boldsymbol{BL}}$.

Lemma 2.4.11. For all ϕ and $\psi \in \mathcal{T}_{\overline{BL^+}}(X \cup X')^\beta$,

$$\vdash_{\mathbf{CF}^+} \phi^L, \psi^R \iff \vdash_{\mathbf{CF}^+_{\boldsymbol{BL}}} \phi^L, \psi^R$$

As a reminder, $\mathbf{CF}^+_{\boldsymbol{BL}}$ is the cut-free proof system for bounded lattices defined in Definition 2.2.14.

Proof. By Lemma 2.4.9, we only need to show $\vdash_{\mathbf{CF}_{\boldsymbol{\delta}}^+} \phi^L, \psi^R \iff \vdash_{\mathbf{CF}_{\boldsymbol{BL}}^+} \phi^L, \psi^R$. As a reminder, the key difference between $\mathbf{CF}_{\boldsymbol{\delta}}^+$ and $\mathbf{CF}_{\boldsymbol{BL}}^+$ is that $\mathbf{CF}_{\boldsymbol{\delta}}^+$ contains the Replace rule.

The backward direction is trivial, as $\mathbf{CF}_{\boldsymbol{BL}}^+$ is a subsystem of $\mathbf{CF}_{\boldsymbol{\delta}}^+$. For the forward direction, the proof proceeds by induction first on the size of the proof tree of $\vdash_{\mathbf{CF}_{\boldsymbol{\delta}}^+} \phi^L, \psi^R$.

The rules Hyp, LeftBot, RightTop, LeftOr, RightOr, LeftAnd and RightAnd of $\mathbf{CF}_{\boldsymbol{\delta}}^+$ also exist in $\mathbf{CF}_{\boldsymbol{BL}}^+$. The remaining case is Replace. Suppose that the replaced term is ϕ^L (a dual argument handles the case of a right term.):

$$\dfrac{\dfrac{\mathcal{A}}{\phi^L, \phi^L}}{\phi^L, \Gamma}\ \textsc{Replace}$$

Note that if $\vdash_{\mathbf{CF}_{\boldsymbol{\delta}}^+} \phi^L, \phi^L$, then $\phi \sim_{OL^+} \bot$. Hence, by Lemma 2.4.10, $\phi \sim_{BL^+} \bot$. Then by completeness (Theorem 2.2.15) there exists a proof of $\vdash_{\mathbf{CF}_{\boldsymbol{BL}}^+} \phi^L, \mathrm{NNF}(\neg\phi)^R$.

□

Corollary 2.4.12. $[\mathcal{T}_{\overline{BL^+}}(X \cup X')^\beta]_{\dashv\vdash_{\mathbf{CF}_{\boldsymbol{BL}}^+}}$ is isomorphic to the free OL^+.

Theorem 2.4.13 (Normalization for OL^+)**.** OL^+ admits a normalization function computable in quadratic time.

Proof. This normalization function is the composition of all the individual simplification steps, defined by $\mathrm{NF}_{OL^+}(\phi) = \eta(\zeta(\beta(\mathrm{NNF}(\phi))))$. Indeed, let $\phi, \psi \in \mathcal{T}_{OL^+}(X)$ such that $\phi \sim_{OL^+} \psi$. Then,

$$\beta(\mathrm{NNF}(\phi)) \sim_{OL^+} \beta(\mathrm{NNF}(\psi))$$

which by Lemma 2.4.11 implies

$$\beta(\mathrm{NNF}(\phi)) \sim_{BL^+} \beta(\mathrm{NNF}(\psi))$$

And then by Theorem 2.4.6 this implies

$$\eta(\zeta(\beta(\mathrm{NNF}(\phi)))) = \eta(\zeta(\beta(\mathrm{NNF}(\psi))))$$

Additionally, $\eta(\zeta(\beta(\mathrm{NNF}(\phi)))) \sim_{OL^+} \phi$ since each step preserves equivalence in OL^+. Finally, we show that the output is in minimal form, when we measure the size of formulas so that negation nodes above variables and function symbols are not counted (in particular, positive and negative literals are both counted to have size 1). For $\phi \in \mathcal{T}_{OL^+}(X)$, let ψ be minimal in the equivalence class of ϕ. Since all of NNF, β, ζ and η can only reduce the size of a formula (not counting negation nodes), $\|\mathrm{NF}_{OL^+}(\psi)\| \leq \|\psi\|$. But since ψ is minimal, $\|\mathrm{NF}_{OL^+}(\psi)\| = \|\psi\|$. Since $\mathrm{NF}_{OL^+}(\phi) = \mathrm{NF}_{OL^+}(\psi)$, $\mathrm{NF}_{OL^+}(\phi)$ is necessarily minimal.

Regarding the time complexity, note that NNF can be computed in linear time and that β takes quadratic time plus the time needed to compute $\leq_{BL^+}$. Note that all formulas that appear during the evaluation of $\eta(\zeta(\beta(\mathrm{NNF}(\phi))))$, are (the normal forms of) subformulas of ϕ or negations of such formulas, so that we indeed need only quadratic time to compute the $\leq_{BL^+}$ relation on all the necessary pairs.

Note that, as in Subsection 2.3.1, we need to use equality of formula nodes through their unique identifiers rather than structural equality to ensure that memoization is effective, and additionally we may not create new nodes in memory during the run of the algorithm. In particular, we must once again use the function `getInverse` from Listing 2.3.

Ultimately, we only ever need to call getInverse, NNF, β, ζ and η on up to $2n$ different inputs and $\leq$ on up to $4n^2$ different inputs. Hence, as long as these functions are properly memoized, we obtain a final quadratic running time.

□

2.5 Interpolation in orthologic

Interpolation-based techniques are important in hardware and software model checking [64, 62, 63, 50, 51, 56, 82]. The interpolation theorem for classical propositional logic states that, for two formulas ϕ and ψ such that $\phi \implies \psi$ is valid, there exists a formula θ, with free variables among only those common to both ϕ and ψ, such that both $\phi \implies \theta$ and $\theta \implies \psi$ are valid. All known algorithms for proof search in classical propositional logic have worst-case exponential size of proofs they construct, which is not surprising given that the validity problem is co-NP-complete. Interpolation algorithms efficiently construct interpolants from such exponentially-sized proofs [79], which makes the overall process exponential. In orthologic, however, since we can find proofs of polynomial size, we should expect to be able to find interpolants of polynomial size as well. That orthologic admits the interpolation property was already shown in [68]. In this section, we give an algorithm to compute orthologic interpolants in polynomial time, using the sequent-based calculus **LO** (without function symbols, Definition 2.2.2).

Acknowledgement of contributions All the contributions of this section are the result of joint work with Sankalp Gambhir.

We consider the following notion of interpolation, which is natural in any lattice-based logic and equivalent in classical logic to Craig interpolation [23].

Definition 2.5.1. Given two formulas ϕ and $\psi \in \mathcal{T}_{OL}(X)$ such that $\phi \leq_{OL} \psi$, an interpolant θ is a formula such that $\phi \leq_{OL} \theta$, $\theta \leq_{OL} \psi$, and $\mathsf{FV}(\theta) \subseteq \mathsf{FV}(\phi) \cap \mathsf{FV}(\psi)$.

We now prove that the theory of ortholattices admits this form of interpolation by showing a procedure constructing the interpolant inductively from a proof of the sequent ϕ^L, ψ^R. For induction, we prove a slightly more general statement:

Theorem 2.5.2 (Interpolant for orthologic sequents)**.** There exists an algorithm (`interpolate` in Listing 2.10), which, given a proof of a sequent Γ, Δ, computes a formula θ, called an *interpolant* for the ordered pair (Γ, Δ), such that $\mathsf{FV}(\theta) \subseteq \mathsf{FV}(\Gamma) \cap \mathsf{FV}(\Delta)$ and the sequents Γ, θ^R and θ^L, Δ are provable. The algorithm has runtime linear in the size of the given proof of Γ, Δ.

Note that interpolants for (Γ, Δ) and for (Δ, Γ) are distinct. In fact, they are negations of each other. Note also that if the sequent Γ, Δ is provable, then by Theorem 2.2.12 there is a proof of it of size at most quadratic in the size of the input sequent. Hence, the interpolation algorithm runs in worst-case time quadratic in the sizes of Γ and Δ.

Proof. We show correctness of the algorithm of Listing 2.10 with inputs Γ, Δ, P where P is a proof of the sequent $S = \Gamma, \Delta$ in the proof system **CF** without axioms, and hence cut-free. We show that the result of `interpolate`(Γ, Δ, P) is an interpolant for (Γ, Δ).

We first deal with the particular case where $\Gamma = \Delta$, as it will simplify the rest of the proof to assume that they are distinct.

Listing 2.10: The algorithm `interpolate` produces an interpolant for any valid sequent, given a proof of this sequent.

```
def interpolate(
  Γ: AnnotatedFormula,
  Δ: AnnotatedFormula,   // the input sequent Γ,Δ
    p: ProofStep         // proof of validity of the input sequent
): Formula =
  if Γ == Δ then Bot //or Top
  else p match
    case Hypothesis() ⇒
      Γ match
        case φ^L ⇒ φ
        case φ^R ⇒ ¬φ
    case Replace((`Γ`, `Γ`), p') ⇒
      Bot
    case Replace((`Δ`, `Δ`), p') ⇒
      Top
    case LeftAnd((φ^L, `Δ`), p') ⇒                          // Γ = (φ ∧ ψ)^L
      interpolate(φ^L, Δ, p')
    case LeftAnd((`Γ`, φ^L), p') ⇒                          // Δ = (φ ∧ ψ)^L
      interpolate(Γ, φ^L, p')
    case LeftOr((φ^L, `Δ`), p₁, (ψ^L, `Δ`), p₂) ⇒          // Γ = (φ ∨ ψ)^L
      interpolate(φ^L, Δ, p₁) ∨ interpolate(ψ^L, Δ, p₂)
    case LeftOr((`Γ`, φ^L), p₁, (`Γ`, ψ^L), p₂) ⇒          // Δ = (φ ∨ ψ)^L
      interpolate(Γ, φ^L, p₁) ∧ interpolate(Γ, ψ^L, p₂)
    // RightAnd and RightOr are dual
    case LeftNot((φ^R, `Δ`), p') ⇒                          // Γ = ¬φ^L
      interpolate(φ^R, Δ, p')
    case LeftNot((`Γ`, φ^R), p') ⇒                          // Δ = ¬φ^L
      interpolate(Γ, φ^R, p')
    ... //Dual cases for Right rules
```

- Suppose $(\Gamma, \Delta) = (\Pi, \Pi)$. Then any formula ψ (in particular $\bot$ and $\top$) is an interpolant as both Π, ψ^R and ψ^L, Π are provable using Replace.

In all other cases, the algorithm works recursively on the proof tree of P, starting from the concluding (root) step. At every step, the algorithm reduces the construction of the interpolant of S to those of its premises. By induction, assume that for a given proof P, the algorithm is correct for all proofs of smaller size (and in particular for the premises of P) and consider every proof step from **CF** (Definition 2.2.9) with which P can be concluded:

- Hyp: suppose the concluding step is

$$\frac{}{\phi^L, \phi^R}\ \textsc{Hyp}$$

We must have $(\Gamma, \Delta) = (\phi^L, \phi^R)$, or $(\Gamma, \Delta) = (\phi^R, \phi^L)$. Assuming the former, $\theta = \phi$ is an interpolant. For the latter, $\theta = \neg\phi$ is an interpolant.

- Replace: suppose the final inference is

$$\frac{\Pi, \Pi}{\Pi, \Sigma}\ \textsc{Replace}$$

As before, we must have $(\Gamma, \Delta) = (\Pi, \Sigma)$ or $(\Gamma, \Delta) = (\Sigma, \Pi)$. In the former case, $\bot$ is an interpolant, as both

$$\frac{\Gamma, \Gamma}{\Gamma, \bot^R}\ \textsc{Replace} \qquad \text{and} \qquad \frac{}{\bot^L, \Delta}\ \textsc{LeftBot}$$

are provable.

The case $\Delta = \Pi$ is analogous, with $\theta = \top$.

- LeftAnd: the proof is of the form

$$\frac{\phi^L, \Pi}{(\phi \wedge \psi)^L, \Pi}\ \textsc{LeftAnd}$$

We must have $(\Gamma, \Delta) = ((\phi \wedge \psi)^L, \Pi)$ or swapped. In the former case, by the induction hypothesis, consider an interpolant C for (ϕ^L, Π), such that the sequents

$$\phi^L, C^R \qquad\qquad C^L, \Delta$$

are provable. Hence, $\theta = C$ is an interpolant, as we have the following proofs:

$$\frac{\phi^L, C^R}{(\phi \wedge \psi)^L, C^R}\ \textsc{LeftAnd} \qquad \text{and} \qquad C^L, \Pi$$

and $\mathsf{FV}(\phi) \subseteq \mathsf{FV}(\phi \wedge \psi)$. The case where $(\Gamma, \Delta) = (\Pi, (\phi \wedge \psi)^L)$ is analogous.

- LeftOr: suppose the final inference is

$$\frac{\phi^L, \Pi \qquad \psi^L, \Pi}{(\phi \vee \psi)^L, \Pi}\ \textsc{LeftOr}$$

We have $(\Gamma, \Delta) = ((\phi \vee \psi)^L, \Pi)$, or the other way round. Assume the former. Applying the induction hypothesis twice, we obtain an interpolant for each of the premises, C_ϕ and C_ψ, such that the sequents

$$\begin{array}{ll} \phi^L, C_\phi^R & \qquad C_\phi^L, \Delta \\ \psi^L, C_\psi^R & \qquad C_\psi^L, \Delta \end{array}$$

are provable. Take $\theta = C_\phi \vee C_\psi$ as an interpolant. Indeed, its free variables are contained in $\mathsf{FV}(\Pi) \cap (\mathsf{FV}(\phi) \cup \mathsf{FV}(\psi)) = \mathsf{FV}(\Pi) \cap \mathsf{FV}(\phi \vee \psi)$.

We then need proofs for Γ, θ^R and θ^L, Δ:

$$\frac{\dfrac{\phi^L, C_\phi^R}{\phi^L, (C_\phi \vee C_\psi)^R}\ \textsc{RightOr} \qquad \dfrac{\psi^L, C_\psi^R}{\psi^L, (C_\phi \vee C_\psi)^R}\ \textsc{RightOr}}{(\phi \vee \psi)^L, (C_\phi \vee C_\psi)^R}\ \textsc{LeftOr}$$

and

$$\frac{C_\phi^L, \Delta \qquad C_\psi^L, \Delta}{(C_\phi \vee C_\psi)^L, \Delta}\ \textsc{LeftOr}$$

showing that θ is an interpolant for the pair (Γ, Δ).

Now in the other case $(\Gamma, \Delta) = (\Pi, (\phi \vee \psi)^L)$, the induction hypothesis gives us the following interpolants:

$$\begin{array}{ll} \Gamma, D_\phi^R & \qquad D_\phi^L, \phi^L \\ \Gamma, D_\psi^R & \qquad D_\psi^L, \psi^L \end{array}$$

We can then take $\theta = (D_\phi \wedge D_\psi)$ to obtain proofs of Γ, θ^R and θ^L, Δ:

$$\frac{\Gamma, D_\phi^R \qquad \Gamma, D_\psi^R}{\Gamma, (D_\phi \wedge D_\psi)^R}\ \textsc{RightAnd}$$

and

$$\frac{\dfrac{D_\phi^L, \phi^L}{(D_\phi \wedge D_\psi)^L, \phi^L}\ \textsc{LeftAnd} \qquad \dfrac{D_\psi^L, \psi^L}{(D_\phi \wedge D_\psi)^L, \psi^L}\ \textsc{LeftAnd}}{(D_\phi \wedge D_\psi)^L, (\phi \vee \psi)^L}\ \textsc{LeftOr}$$

Note that it is easy to see by induction that $D_\phi \wedge D_\psi = \neg(C_\phi \vee C_\psi)$.

- LeftNot: suppose the final inference is

$$\frac{\Pi, \phi^R}{\Pi, \neg\phi^L}\ \textsc{LeftNot}$$

We have $(\Gamma, \Delta) = (\Pi, (\neg\phi)^L)$, or the other way round. Assume the former. We apply the induction hypothesis as before to obtain an interpolant C for (Γ, ϕ^R) such that

$$\Gamma, C^R \qquad\qquad C^L, \phi^R$$

are provable. $\theta = C$ suffices as an interpolant for the concluding sequent, with the proofs of Γ, θ^R and θ^L, Δ

$$\Gamma, C^R \qquad \text{and} \qquad \frac{C^L, \phi^R}{C^L, \neg\phi^L}\ \textsc{LeftNot}$$

The proofs for the remaining proof rules, RightAnd, RightOr, and RightNot, are analogous to the cases listed above. Hence, Listing 2.10 correctly computes an interpolant of Γ, Δ. □

Corollary 2.5.3 (Interpolation for ortholattices)**.** Orthologic admits interpolation, i.e., for any pair of formulas $\phi, \psi \in \mathcal{T}_{OL}(X)$ with $\phi \leq \psi$, there exists a formula θ such that $\phi \leq \theta$ and $\theta \leq \psi$, with $\mathsf{FV}(\theta) \subseteq \mathsf{FV}(\phi) \cap \mathsf{FV}(\psi)$.

2.6 Proof strength of orthologic with axioms

Section 2.2 established a cubic-time algorithm (Theorem 2.2.12) for deriving all consequences of axioms that hold in orthologic. This generalization is sound for classical logic while still being efficient. A natural question then is: how precise is it as an approximation of classical logic?

In this section, we present several classes of classical propositional problems that are solved *exactly* by orthologic. *OL* proof search is then not only sound for them (as it is for all problems of classical logic), but also *complete*: it always finds a proof if, e.g., a SAT solver would find it.

We are interested in traditional classes of deduction problems that are solvable by *OL* proofs. Formally, we define the deduction problem in orthologic and, respectively, in classical logic.

Definition 2.6.1. An instance of the deduction problem is characterized by a pair (A, S) where A is a set of axioms and S is the goal, all of which are sequents whose interpretation as an inequality is given by Definition 2.2.4. The deduction problem in *OL* (resp. *CL*) consists in deciding if the goal S can be derived from axioms A in orthologic (resp. classical logic). If this is the case, then the instance is called *valid*.

Definition 2.6.2. An instance of the deduction problem is *OL*-solvable if and only if it has the same validity in *OL* and *CL*. A class of instances of the deduction problem is *OL*-solvable if and only if all its members are *OL*-solvable.

As *OL* is sound relative to *CL*, the following are equivalent formulations of *OL*-solvability:

- If the instance has a proof in *CL* then it has a proof in *OL*.
- If the goal of the instance is true in all $\{\bot, \top\}$ interpretations satisfying the axioms, then it is true in all ortholattice interpretations satisfying the axioms.

If the goal of the instance is the empty sequent (that is, $\bot^R, \bot^R$ in Definition 2.2.2), the validity of the instance corresponds to the inconsistency (unsatisfiability) of the axioms.

Formally, *OL*-solvability of $(A, \emptyset)$ is equivalent to each of the following statements:

- The axioms of the instance are either unsatisfiable in *OL* or satisfiable in *CL*.
- The axioms of the instance either have a non-trivial Boolean model, or admit only the trivial one-element structure as a model among all ortholattices.

In particular, Theorem 2.2.12 gives a polynomial-time decision procedure for the satisfiability problem with respect to classical logic for any class of deduction problem instances that are *OL*-solvable.

We look at the satisfiability of propositional logic formulas in conjunctive normal form (CNF), which are conjunctions of clauses, as their analysis plays an important role in proof theory of *CL* and the practice of SAT solving.

Among the simplest and most studied refutationally complete systems for *CL* is resolution on clauses, shown in Figure 2.2.

$$\frac{C, x \qquad C', \neg x}{C, C'}\ \textsc{Resolution}$$

$$\frac{}{C, x, \neg x}\ \textsc{Hyp} \qquad \frac{C}{C, C'}\ \textsc{Weakening}$$

Figure 2.2: The Resolution proof system with hypothesis and weakening rules. C and C' represent arbitrary sets of literals. This system is complete for deriving contradictions in CL, with formulas expressed in conjunctive normal form, even without the Weakening and Hyp rules [81, Chapter 2].

Definition 2.6.3. A *literal* is either a variable symbol or the negation of a variable. A *clause* is a set of literals. The *Resolution* proof system (for classical logic) is shown in Figure 2.2.

2.6.1 Completeness for 2SAT

We start with the simplest example of a CNF, the 2CNF class. A 2CNF formula is a finite set of clauses $C_1, \ldots, C_m$, where each clause is a disjunction of two literals or a single literal. For example, $(x \vee \neg y), (\neg x \vee z), (\neg z)$ is a 2CNF formula. 2SAT is the problem of deciding if a 2CNF formula is satisfiable, i.e. if it has a model in the two-element Boolean algebra. Conversely, the instance is unsatisfiable if and only if the conjunction of the clauses implies falsity.

We can encode a 2SAT instance into an OL deduction problem $(A, \emptyset)$ where each axiom in A is a sequent containing at most two labelled *variables* as formulas. We create an axiom sequent for each clause, where a negative literal $\neg p$ becomes a labelled formula p^L and a positive literal p becomes p^R. For example, $\{\neg x, y\}$ becomes x^L, y^R. Similarly, $\{x, y\}$ becomes x^R, y^R, whereas $\{x\}$ becomes x^R. This encoding is equivalent (in CL) to the 2SAT instance, with the interpretation of sequents given in Definition 2.2.4.

Consider the Resolution proof system shown in Figure 2.2. For 2SAT instances, the outcome of resolution can be simulated by orthologic using theCut rule, which allows us to prove the following.

Theorem 2.6.4 (2SAT completeness)**.** 2SAT is OL-solvable.

Proof. Consider a 2CNF problem and its representation as a set of sequents. The instance is unsatisfiable in CL if and only if there exists a derivation of the empty clause in Resolution. We proceed by induction on the Resolution derivation to show that if a clause is derived, then there is an OL-proof of the corresponding sequent.

Consider a Resolution step between two clauses of (at most) two elements. The sequent corresponding to its conclusion can be deduced from the sequent corresponding to its premises by a single application of the Cut rule in orthologic.

$$\frac{\{\gamma, y\} \qquad \{\neg y, \delta\}}{\gamma, \delta}\ \textsc{Resolution} \quad \hookrightarrow \quad \frac{\{\Gamma, y^R\} \qquad \{y^L, \Delta\}}{\Gamma, \Delta}\ \textsc{Cut}$$

where γ and δ are arbitrary literals and Γ and Δ represent the corresponding annotated formulas. WEAKENING is similarly simulated by the REPLACE rule of **LO**, and HYP by the eponymous step.

Conversely, if the empty sequent is derivable in orthologic, then no non-trivial ortholattice satisfies the assumptions, and hence no Boolean algebra does either. □

2.6.2 Orthologic emulates unit resolution

For an arbitrary clause $N \cup P$, let it be encoded as the sequent $(\bigwedge N)^L, (\bigvee P)^R$, where N (resp. P) is the set of negative (resp. positive) variables in the clause. A *Unit Resolution* step is a Resolution step (Figure 2.2) where $C = \emptyset$ or $C' = \emptyset$. Interpreted over sequents, the application of a Unit Resolution step on two clauses $C_1 = \{x_i\}$ and $C_2 = (\{\neg x_1, \ldots, \neg x_n\} \cup P)$ corresponds to the deduction rule UNITRESOLUTIONR. Dually, for $C_1 = \{\neg x_i\}$ we obtain UNITRESOLUTIONL:

$$\frac{(x_i)^R \qquad (x_1 \wedge \ldots \wedge x_{i-1} \wedge x_i \wedge x_{i+1} \wedge \ldots \wedge x_n)^L, (\bigvee P)^R}{(x_1 \wedge \ldots \wedge x_{i-1} \wedge x_{i+1} \wedge \ldots \wedge x_n)^L, (\bigvee P)^R}\ \textsc{UnitResolutionR}$$

$$\frac{(x_i)^L \qquad (\bigwedge N)^L, (x_1 \vee \ldots \vee x_{i-1} \vee x_i \vee x_{i+1} \vee \ldots \vee x_n)^R}{(\bigwedge N)^L, (x_1 \vee \ldots \vee x_{i-1} \vee x_{i+1} \vee \ldots \vee x_n)^R}\ \textsc{UnitResolutionL}$$

Lemma 2.6.5 (*OL* simulates unit resolution)**.** UNITRESOLUTIONL and UNITRESOLUTIONR are admissible rules in **LO** (Definition 2.2.2).

Proof. We aim to show that this step is admissible in *OL* proofs. Instead of giving a syntactic transformation, we can see that the steps are sound in *OL* with a short semantic argument. For UNITRESOLUTIONR, consider any ortholattice and assume that the premises of the rule are true. The interpretation of x_i^R is $\top \leq x_i$, which implies $x_i = \top$. This implies $(x_1 \wedge \ldots \wedge x_i \wedge \ldots \wedge x_n) = (x_1 \wedge \ldots \wedge \top \wedge \ldots \wedge x_n)$, so the value of the non-unit premise clause reduces to the truth of the conclusion of the rule. By completeness (Theorem 2.2.6), there exists a proof of the conclusion from the premises. Thus, UNITRESOLUTIONR is admissible. The argument for UNITRESOLUTIONL is dual. □

Completeness for Horn clauses

A Horn clause is a disjunction of literals such that at most one literal is positive. We encode a Horn clause into a sequent as $(a_1 \wedge \ldots \wedge a_n)^L, b^R$ where a_i are the negated variables of the clause (if any), and b is the positive literal of the clause (if it exists). A Horn instance is a conjunction of Horn clauses, and is unsatisfiable if and only if the empty clause can be deduced from it using Resolution.

We can again encode a Horn instance into a deduction problem by adding an axiom for each Horn clause, and the empty sequent as the goal.

Horn instances can be solved using only Unit Resolution [67]. By Lemma 2.6.5, *OL* is complete for Horn instances. Other classes solvable using only Unit Resolution include *q-Horn*, *extended Horn* and *renamed Horn* instances [20].

Corollary 2.6.6 (Horn clauses completeness)**.** Horn clause instances, q-Horn instances, extended Horn instances and renamed Horn instances are *OL*-solvable classes.

Note that, despite our use of semantic techniques to show completeness for resolution, the results of Theorem 2.2.12 apply and provide polynomial-time guarantees for solving these instances. In Section 2.2 we used Horn clauses to show that orthologic proof search is polynomial-time, but note that this is coincidental and unrelated to the present result. In Section 2.12, we will present a different algorithm for the entailment problem in *OL* that does not use Horn clauses.

Renamed Horn instances are an interesting extension of Horn instances. A conjunction of clauses is renamed Horn if and only if there exists a set of variables V such that complementing variables of V in the instance yields a Horn instance. In particular, a clause in a renamed Horn instance can contain multiple positive and negative literals. Unit Resolution is stable under such renaming, meaning that a unit resolution derivation of the empty clause remains a valid unit resolution derivation of the empty clause after renaming. This is the reason that Unit Resolution is also complete for renamed Horn clauses.

2.6.3 Renaming deduction problems

Motivated by renamed Horn instances, we now consider renaming of general deduction problems. We show that renamings of *OL*-solvable instances are *OL*-solvable.

Definition 2.6.7. For an arbitrary variable x, the complement of x is $\neg x$ and the complement of $\neg x$ is x. Two deduction problems I_1 and I_2 are *renamings* of each other if there exists a set of variables V such that complementing variables of V in the axioms and goal of I_1 yields I_2.

Lemma 2.6.8. Let I be a deduction instance such that I is *OL*-solvable. Then all renamed versions of I are *OL*-solvable.

Proof. We first show that, in *OL*, if I_1 and I_2 are renamed versions of each other by a set of variables V, I_1 and I_2 have the same validity. Indeed, suppose there is an ortholattice $\mathcal{O}$ and assignment $s_1 : V \to \mathcal{O}$ that is a countermodel of I_1 (meaning it satisfies the axioms but not the goal, so I_1 is invalid). Then define s_2 such that $s_2(x) = s_1(x)$ if $x \notin V$ and $s_2(x) = \neg s_1(x)$ if $x \in V$. It is then easy to check that since $\neg\neg x = x$ in *OL*, s_2 is a countermodel for I_2. Hence if I_1 is invalid then I_2 is invalid. Conversely, if I_2 admits a countermodel, then I_1 admits a countermodel as well.

Similarly in *CL*, I_1 and I_2 have the same validity by the same argument, considering assignments $V \to \{\bot, \top\}$ as countermodels. Since I_1 is *OL*-solvable, then

$$I_2 \text{ is valid in } OL \iff I_1 \text{ is valid in } OL \iff I_1 \text{ is valid in } CL \iff I_2 \text{ is valid in } CL$$

and hence I_2 is *OL*-solvable.

□

2.6.4 Tseitin transformation for orthologic axioms

Tseitin transformation for classical logic transforms a formula with arbitrary alternations of conjunctions and disjunctions into one in Conjunctive Normal Form (CNF) that is equisatisfiable. The transformation works by introducing a linear number of new variables, and runs in near linear time. In practice, most SAT solvers work on formulas in CNF.

The essence of Tseitin transformation in classical logic is to introduce variables, such as x, that serve as names for subformulas, such as F, with the interpretation of x bound to be equal to the interpretation of F. Unfortunately, we cannot express such a transformation as a single *OL* formula, because knowing that $\neg x \vee F$ equals $\top$ inside a formula does not imply $x \leq F$. On the other hand, once we make use of the power of axioms, the usual Tseitin transformation again becomes possible.

Definition 2.6.9 (*OL*-Tseitin transformation)**.** Given a deduction problem instance with axioms A and goal S (where we assume without loss of generality that all formulas are in negation normal form), pick an arbitrary strict subexpression e of a formula in A or S of the form $x \wedge y$ or $x \vee y$, for some literals x and y. Pick a fresh variable c, introduce the axioms c^L, e^R and e^L, c^R and replace e by c in A and S. Repeat until all formulas have height at most 2. We say that a problem (A, S) is in Tseitin normal form if it is obtained from another problem using this transformation.

Theorem 2.6.10 (Validity of *OL*-Tseitin transformation)**.** An instance of the deduction problem is valid if and only if its *OL*-Tseitin transformation is valid.

Proof. Consider a counter-assignment to the original problem. It can be extended to a counter-assignment to the transformed problem by assigning to each fresh variable c the same value as the subexpression e it replaces. Conversely, assume the transformed problem has a counter-assignment. By the axioms c^L, e^R and e^L, c^R, c and e must have the same value in the counter-assignment. Hence, restricting the assignment to the original variables yields a counter-assignment to the original problem. □

2.6.5 Resolution width for orthologic proofs in CNF

Consider an *OL* proof for a deduction problem to which we applied *OL*-Tseitin transformation. With $\circ$ and $\square$ denoting arbitrary $\_^L$ or $\_^R$ annotations, the resulting problem will only contain sequents of the form: $\{a^\circ, (b \wedge c)^\square\}$, $\{a^\circ, (b \vee c)^\square\}$, for some literals a, b and c. Moreover, remember that by the subformula property (Theorem 2.2.11), if the problem admits a proof, then it has a proof that only uses formulas among a, $a \wedge b$, and $a \vee b$ (for any literals a and b appearing in the problem). Hence, we can constrain every proof of S to involve only sequents of at most 4 literals. In classical logic, every sequent appearing in the proof would then be equivalent to a conjunction of disjunctions of literals (i.e. a conjunction of clauses). In the simplest case:

$$(w \wedge x)^L, (y \vee z)^R \quad \rightsquigarrow \quad (\neg w \vee \neg x \vee y \vee z)$$

For sequents involving a left disjunction or right conjunction, using greatest lower bound and least upper bound properties of $\wedge$ and $\vee$:

$$\begin{aligned}
(w \vee x)^L, (y \vee z)^R &\quad\rightsquigarrow\quad (\neg w \vee y \vee z) \wedge (\neg x \vee y \vee z) \\
(w \wedge x)^L, (y \wedge z)^R &\quad\rightsquigarrow\quad (\neg w \vee \neg x \vee y) \wedge (\neg w \vee \neg x \vee z) \\
(w \vee x)^L, (y \wedge z)^R &\quad\rightsquigarrow\quad (\neg w \vee y) \wedge (\neg w \vee z) \wedge (\neg x \vee y) \wedge (\neg x \vee z)
\end{aligned}$$

Similarly, with all combinations of ^{L} and ^{R}, where, for example, a conjunction with L polarity behaves much like a disjunction with R polarity. We say that such a set of clauses *represents* the corresponding sequent. Crucially, each of these clauses contains at most 4 literals.

We now consider again the Resolution proof system of Figure 2.2. Note that we work with plain resolution and *not* extended resolution that introduces fresh variables on the fly [93].

Definition 2.6.11. The **width** of a resolution proof is the number of literals in the largest clause appearing in the proof.

The next theorem characterizes the width of those classical logic resolution proofs that suffice to establish all formulas provable by *OL* derivations.

Theorem 2.6.12 (Resolution width bound)**.** An **LO** proof of a problem in *OL*-Tseitin normal form can be simulated by Figure 2.2 proofs of width 5.

Proof. Consider an **LO** proof of a problem in *OL*-Tseitin normal form. We proceed by induction on the structure of such a proof. The cases of non-Cut rules are immediate. Namely, Hyp in **LO** is directly simulated by the corresponding steps in Resolution, and Replace is simulated by Weakening. In LeftNot, the principal formula must be a literal, so that the clause representation of the conclusion is the same as the clause representation of the premise. In LeftOr, similarly, the representation of the conclusion is the conjunction of the representations of the premises. LeftAnd is simulated in a Resolution proof by Weakening. Right- steps are symmetric to Left- steps. The only non-trivial case is the Cut rule. The cut formula can have different shapes

1. The cut formula is a literal

$$\frac{\Gamma, x^R \qquad x^L, \Delta}{\Gamma, \Delta}\ \textsc{Cut}$$

   We then have technically 36 different cases to describe depending on whether Γ and Δ are conjunctions, disjunctions or literals and their polarity. We show the two extreme cases, and all others can be deduced by symmetry.

   If Γ and Δ are left conjunctions, right disjunctions, or literals, the Cut rule is simulated with a single Resolution instance:

$$\dfrac{\dfrac{(a \wedge b)^L, x^R \qquad x^L, (c \vee d)^R}{\Gamma, \Delta}\,\textsc{Cut}}{} \quad \hookrightarrow \quad \dfrac{\dfrac{\{\neg a, \neg b, x\} \qquad \{\neg x, c, d\}}{\{\neg a, \neg b, c, d\}}\,\textsc{Resolution}}{}$$

If Γ and Δ are left disjunctions or right conjunctions, 2 applications of Resolution are necessary to obtain each of the two clauses in the conclusion (4 if both):

$$\dfrac{(a \vee b)^L, x^R \qquad x^L, (c \wedge d)^R}{\Gamma, \Delta}\,\textsc{Cut}$$

$$\hookrightarrow$$

$$\dfrac{\{\neg a, x\}, \{\neg b, x\} \qquad \{\neg x, c\}, \{\neg x, d\}}{\{\neg a, c\}, \{\neg a, d\}, \{\neg b, c\}, \{\neg b, d\}}\,4\times\ \textsc{Resolution}$$

where each of the clauses in the conclusion can be reached by using Resolution on two of the clauses in the premises.

2. Consider now the case where the Cut formula is a conjunction (the disjunction case is symmetric).

$$\dfrac{\Gamma, (x \wedge y)^R \qquad (x \wedge y)^L, \Delta}{\Gamma, \Delta}\,\textsc{Cut}$$

We again present the two extreme cases. If Γ and Δ are both left conjunctions or right disjunctions:

$$\dfrac{(a \wedge b)^L, (x \wedge y)^R \qquad (x \wedge y)^L, (c \vee d)^R}{(a \wedge b)^L, (c \vee d)^R}\,\textsc{Cut}$$

$$\hookrightarrow$$

$$\dfrac{\dfrac{\{\neg a, \neg b, x\},\ \{\neg a, \neg b, y\} \qquad \{c, d, \neg x, \neg y\}}{\{\neg a, \neg b, c, d, \neg y\}}\,\textsc{Resolution on } x}{\{\neg a, \neg b, c, d\}}\,\textsc{Resolution on } y$$

The conclusion can be reached by applying Resolution twice successively, but here the intermediate clause $\{\neg a, \neg b, c, d, \neg y\}$ reaches width 5.

If Γ and Δ are left disjunctions or right conjunctions:

$$\dfrac{(a \vee b)^L, (x \wedge y)^R \qquad (x \wedge y)^L, (c \wedge d)^R}{(a \vee b)^L, (c \wedge d)^R}\,\textsc{Cut}$$

$$\hookrightarrow$$

$$\dfrac{\dfrac{\{\neg a, x\},\ \{\neg a, y\}, \{\neg b, x\},\ \{\neg b, y\} \qquad \{\neg x, \neg y, c\}, \{\neg x, \neg y, d\}}{\{\neg y, \neg a, c\}, \{\neg y, \neg a, d\}, \{\neg y, \neg b, c\}, \{\neg y, \neg b, d\}}\,4\times\ \textsc{Resolution on } x}{\{\neg a, c\}, \{\neg a, d\}, \{\neg b, c\}, \{\neg b, d\}}\,4\times\ \textsc{Resolution on } y$$

then Cut can be simulated by first resolving 4 times on x and then 4 times on y.

Hence, Resolution of width 5 can simulate all *OL* proofs. □

2.7 Quantified orthologic

We consider the extension of propositional orthologic to quantified orthologic, denoted *QOL*, the analogue of QBF [21] for classical logic, or of System F [33] for intuitionistic logic. To do so, we extend the syntax of orthologic and the sequent calculus **LO** by adding the axiomatization of an existential quantifier ($\bigvee$) and a universal quantifier ($\bigwedge$).

Definition 2.7.1. $\mathcal{T}_{QOL}(X)$ denotes the set of quantified orthologic formulas, i.e. $\mathcal{T}_{OL}(X) \subset \mathcal{T}_{QOL}(X)$ and for any $x \in X$ and $\phi \in \mathcal{T}_{QOL}(X)$,

$$\bigwedge x.\ \phi \in \mathcal{T}_{QOL}(X) \quad \text{and} \quad \bigvee x.\ \phi \in \mathcal{T}_{QOL}(X).$$

Note that $\mathcal{T}_{QOL}(X)$ is identical to the set of quantified Boolean formulas. For two formulas $\phi, \psi \in \mathcal{T}_{QOL}(X)$ and a variable x, let $\phi[x := \psi]$ denote the usual capture-avoiding substitution of x by ψ inside ϕ.

We then define the proof system **QLO**, which contains all rules of **LO** as well as the four rules of Figure 2.3. As usual, we write $\vdash_{\mathbf{QLO}} s$ to denote that s has a proof in **QLO**.

QOL differs from the first-order theory of ortholattices. In particular, the semantics of an existential quantifier ($\bigvee x.\ t$) in *QOL* corresponds to the least upper bound of a (possibly infinite) family of *ortholattice* elements given by values of the term t. In contrast, when considering a classical first-order theory of ortholattices, we would build an atomic formula such as $t_1 \leq t_2$, obtaining a definite truth or falsehood in the metatheory, and only then apply quantifiers to build formulas such as $\exists x.(t_1 \leq t_2)$.

Acknowledgement of contributions All the contributions of this section are the result of joint work with Sankalp Gambhir.

2.7.1 Complete ortholattices

To model quantified orthologic, we restrict ortholattices to complete ones.

Definition 2.7.2 (Complete ortholattice)**.** An ortholattice $\mathcal{O} = (O, \sqsubseteq, \sqcup, \sqcap, -)$ is *complete* if and only if for any possibly infinite set of elements $S \subseteq O$, there exist two

$$\frac{\Gamma, \phi[x := \psi]^L}{\Gamma, (\bigwedge x.\ \phi)^L}\ \textsc{LeftForall} \qquad \frac{\Gamma, \phi[x := x']^R}{\Gamma, (\bigwedge x.\ \phi)^R}\ \textsc{RightForall}\ (x' \text{ not free in } \Gamma)$$

$$\frac{\Gamma, \phi[x := x']^L}{\Gamma, (\bigvee x.\ \phi)^L}\ \textsc{LeftExists}\ (x' \text{ not free in } \Gamma) \qquad \frac{\Gamma, \phi[x := \psi]^R}{\Gamma, (\bigvee x.\ \phi)^R}\ \textsc{RightExists}$$

Figure 2.3: Deduction rules of Quantified Orthologic.

elements denoted $\bigsqcup S$ and $\bigsqcap S$ which are respectively the *least upper bound* and *greatest lower bound* of elements of S, with respect to $\sqsubseteq$:

$$\forall x \in S.\ \ x \sqsubseteq \bigsqcup S \text{ and } \bigsqcap S \sqsubseteq x\ ,$$

and, for all $y \in O$:

$$(\forall x \in S. x \sqsubseteq y) \implies (\bigsqcup S \sqsubseteq y)$$

$$(\forall x \in S. y \sqsubseteq x) \implies (y \sqsubseteq \bigsqcap S)$$

This definition coincides with the usual definition of complete lattices. Note that, in particular, all finite ortholattices are complete with bounds computed by iterating the binary operators $\sqcup$ and $\sqcap$.

Definition 2.7.3 (Models and interpretation)**.** A *model* for *QOL* is a complete ortholattice $\mathcal{O} = (O, \sqsubseteq, \sqcup, \sqcap, -)$ and an assignment $\sigma : X \to O$. The interpretation of a formula ϕ with respect to an assignment σ is defined recursively, extending the interpretation of the formulas of propositional orthologic into ortholattices of Definition 2.1.6:

$$\begin{aligned}
[\![x]\!]_\sigma &:= \sigma(x) \\
[\![\phi \wedge \psi]\!]_\sigma &:= [\![\phi]\!]_\sigma \sqcap [\![\psi]\!]_\sigma \\
[\![\phi \vee \psi]\!]_\sigma &:= [\![\phi]\!]_\sigma \sqcup [\![\psi]\!]_\sigma \\
[\![\neg\phi]\!]_\sigma &:= -[\![\phi]\!]_\sigma \\
[\![\bigvee x.\ \phi]\!]_\sigma &:= \bigsqcup\{[\![\phi]\!]_{\sigma[x:=e]} \mid e \in O\} \\
[\![\bigwedge x.\ \phi]\!]_\sigma &:= \bigsqcap\{[\![\phi]\!]_{\sigma[x:=e]} \mid e \in O\}
\end{aligned}$$

where $\sigma[x := e]$ denotes the assignment σ with its value at x changed to e and all other values unchanged.

The interpretation of a sequent is defined as in Definition 2.2.4.

We show soundness and completeness of the **QLO** proof system with respect to the class of all complete ortholattices. Soundness is easy and direct, completeness less so.

Lemma 2.7.4 (Soundness of **QLO**)**.** For every sequent s, if $\vdash_{\mathbf{QLO}} s$ then s holds in every complete ortholattice.

Proof. We simply extend the proof of Theorem 2.2.5 to show that every deduction rule of Figure 2.3 preserves truth of interpretation in any model. We only need to treat the new quantifier rules.

LeftForall: We need to show that for any assignment $\sigma : X \to O$,

$$\left[\!\!\left[\Gamma, (\bigwedge x.\ \phi)^L\right]\!\!\right]_\sigma$$

where we can again assume $\Gamma = \gamma^R$ without loss of generality. Then

$$\begin{aligned} \left[\!\left[\gamma^R, (\bigwedge x.\ \phi)^L\right]\!\right]_\sigma &\iff \\ \left[\!\left[\bigwedge x.\ \phi\right]\!\right]_\sigma \sqsubseteq [\![\gamma]\!]_\sigma &\iff \\ \bigsqcap\{[\![\phi]\!]_{\sigma[x:=e]} \mid e \in O\} \sqsubseteq [\![\gamma]\!]_\sigma & \end{aligned}$$

By induction on the premise of LeftForall, there exists a formula ψ such that $\left[\!\left[\gamma^R, \phi[x := \psi]^L\right]\!\right]_\sigma$ holds true. Finally,

$$\begin{aligned} \left[\!\left[\gamma^R, \phi[x := \psi]^L\right]\!\right]_\sigma &\iff \\ [\![\phi[x := \psi]]\!]_\sigma \sqsubseteq [\![\gamma]\!]_\sigma &\iff \\ [\![\phi]\!]_{\sigma[x:=[\![\psi]\!]_\sigma]} \sqsubseteq [\![\gamma]\!]_\sigma &\implies \\ \bigsqcap\{[\![\phi]\!]_{\sigma[x:=e]} \mid e \in O\} \sqsubseteq [\![\gamma]\!]_\sigma & \end{aligned}$$

where the last implication holds by definition of $\bigsqcap$.

LeftExists: We need to show that for any assignment $\sigma : X \to O$,

$$\left[\!\left[\Gamma, (\bigvee x.\ \phi)^L\right]\!\right]_\sigma$$

where we assume again without loss of generality that $\Gamma = \gamma^R$. Then

$$\begin{aligned} \left[\!\left[\gamma^R, (\bigvee x.\ \phi)^L\right]\!\right]_\sigma &\iff \\ \left[\!\left[\bigvee x.\ \phi\right]\!\right]_\sigma \sqsubseteq [\![\gamma]\!]_\sigma &\iff \\ \bigsqcup\{[\![\phi]\!]_{\sigma[x:=e]} \mid e \in O\} \sqsubseteq [\![\gamma]\!]_\sigma & \end{aligned}$$

By hypothesis, $\left[\!\left[\gamma^R, \phi^L\right]\!\right]_\tau$ holds for any assignment τ, and in particular for any τ of the form $\sigma[x := e]$. Since x does not appear in γ, $[\![\gamma]\!]_{\sigma[x:=e]} = [\![\gamma]\!]_\sigma$. Hence, for any $e \in O$, each of the following lines holds true:

$$\begin{aligned} \left[\!\left[\gamma^R, \phi^L\right]\!\right]_{\sigma[x:=e]} &\iff \\ [\![\phi]\!]_{\sigma[x:=e]} \sqsubseteq [\![\gamma]\!]_{\sigma[x:=e]} &\iff \\ [\![\phi]\!]_{\sigma[x:=e]} \sqsubseteq [\![\gamma]\!]_\sigma & \end{aligned}$$

By the least upper bound property of $\bigsqcup$, we obtain as desired the truth of:

$$\bigsqcup\{[\![\phi]\!]_{\sigma[x:=e]} \mid e \in O\} \sqsubseteq [\![\gamma]\!]_\sigma$$

□

2.7.2 Completeness

In classical propositional logic, we can show completeness with respect to the $\{0, 1\}$ Boolean algebra, which is straightforward. In orthologic, however, we do not have completeness with respect to a simple finite structure; we will need to build an infinite complete ortholattice. The construction is similar to, but distinct from, that of models for predicate orthologic [68, 7]. In particular, we will use the MacNeille completion [59] to transform the initial incomplete model into a complete one.

Lemma 2.7.5 (Completeness)**.** For any sequent s, if s holds in every complete ortholattice then $\vdash_{\mathbf{QLO}} s$.

Proof. We prove the contrapositive: if the sequent s is not provable, then there exists a complete ortholattice $\mathcal{O} = (O, \sqsubseteq, \sqcup, \sqcap, -)$ and an assignment $\sigma : X \to O$ such that $[\![s]\!]_\sigma$ does not hold. We construct O from the set of syntactic terms of complete ortholattices themselves, similarly to a free algebra (but with quantifiers). Formally, let O be $\mathcal{T}_{QOL}(X)/{\sim_{\dashv\vdash}}$, i.e. the quotient set of $\mathcal{T}_{QOL}(X)$ by the relation $\dashv\vdash$.

It is immediate that the function symbols $\wedge, \vee, \neg$ and relation $\vdash_{\mathbf{QLO}}$ of $\mathcal{T}_{QOL}(X)$ are consistent over the equivalence classes of O, allowing us to extend them to O:

$$\begin{aligned}
[\phi]_{\dashv\vdash} \sqcap [\psi]_{\dashv\vdash} &:= [\phi \wedge \psi]_{\dashv\vdash} \\
[\phi]_{\dashv\vdash} \sqcup [\psi]_{\dashv\vdash} &:= [\phi \vee \psi]_{\dashv\vdash} \\
-[\phi]_{\dashv\vdash} &:= [\neg\phi]_{\dashv\vdash} \\
[\phi]_{\dashv\vdash} \sqsubseteq [\psi]_{\dashv\vdash} &:= (\phi \vdash_{\mathbf{QLO}} \psi) \text{ is provable}
\end{aligned}$$

It is also immediate that $\mathcal{O} = (O, \sqsubseteq, \sqcup, \sqcap, -)$ satisfies all the laws of ortholattices of Table 2.1. However, to interpret a quantified formula into $\mathcal{O}$, we would need $\mathcal{O}$ to be complete. It might not be complete, but it is "complete enough" to define all upper bounds of interest, as the following lemma shows.

Lemma 2.7.6. For any $\sigma : X \to O$, let $\sigma' : X \to \mathcal{T}_{QOL}(X)$ be such that for any x $[\sigma'(x)]_{\dashv\vdash} = \sigma(x)$. Let $\phi[\sigma']$ denote the simultaneous capture-avoiding substitution of variables in the formula ϕ with the assignments in σ'.

Then, for any $\phi \in \mathcal{T}_{QOL}(X)$, $[\![\phi]\!]_\sigma$ exists and $[\![\phi]\!]_\sigma = [\phi[\sigma']]_{\dashv\vdash}$.

Proof. First note that $[\phi[\sigma']]_{\dashv\vdash}$ is well-defined: it does not depend on the specific choice of assignment we make for σ'. Then, the proof works by structural induction on ϕ. If it is a variable x,

$$[\![x]\!]_\sigma = \sigma(x) = [\sigma'(x)]_{\dashv\vdash} = [x[\sigma']]_{\dashv\vdash}$$

by definition. Then, if $\phi = \phi_1 \wedge \phi_2$,

$$[\![\phi_1 \wedge \phi_2]\!]_\sigma = [\![\phi_1]\!]_\sigma \sqcap [\![\phi_2]\!]_\sigma = [\phi_1[\sigma']]_{\dashv\vdash} \sqcap [\phi_2[\sigma']]_{\dashv\vdash} = [\phi_1[\sigma'] \wedge \phi_2[\sigma']]_{\dashv\vdash}$$

where the first equality is the definition of $[\![\cdot]\!]$, the second equality is the induction hypothesis and the third equality is the definition of $\sqcap$ in $\mathcal{O}$. $\vee$ and $\neg$ are similar.

Consider now the interpretation of a formula $[\![\bigvee x.\ \phi]\!]_\sigma$. Since alpha-equivalence holds in our proof system and in the definition of the least upper bound, we assume, to ease notation, that x is fresh with respect to σ, i.e., that we don't need to explicitly signal capture-avoiding substitution. By definition of $[\![\cdot]\!]_\sigma$, we should have:

$$\left[\!\!\left[\bigvee x.\ \phi\right]\!\!\right]_\sigma = \bigsqcup\{[\![\phi]\!]_{\sigma[x:=e]} \mid e \in O\}$$

Does the right-hand side always exist in $\mathcal{O}$? We claim that it does, and that it is equal to $[(\bigvee x.\ \phi)[\sigma']]_{\dashv\vdash}$. In particular, we need to show that it satisfies the two properties of the least upper bound. First, the *upper bound* property:

$$\forall a \in \{[\![\phi]\!]_{\sigma[x:=e]} \mid e \in O\},\quad a \sqsubseteq [(\bigvee x.\ \phi)[\sigma']]_{\dashv\vdash}$$

which is equivalent to $\forall e \in O$,

$$[\![\phi]\!]_{\sigma[x:=e]} \sqsubseteq [(\bigvee x.\ \phi)[\sigma']]_{\dashv\vdash} \qquad \iff \qquad (2.2)$$

$$[\phi[\sigma'_{[x:=e]}]]_{\dashv\vdash} \sqsubseteq [(\bigvee x.\ \phi)[\sigma']]_{\dashv\vdash} \qquad \iff \qquad (2.3)$$

$$\phi[\sigma'_{[x:=e]}] \vdash_{\mathbf{QLO}} (\bigvee x.\ \phi)[\sigma'] \text{ is provable} \qquad \iff \qquad (2.4)$$

$$\phi[\sigma'_{[x:=e]}] \vdash_{\mathbf{QLO}} \bigvee x.\ \phi[\sigma'_{[x:=x]}] \text{ is provable} \qquad (2.5)$$

where (1) is the desired least upper bound property, (2) is equivalent by induction hypothesis and definition of σ', (3) by definition of $\sqsubseteq$ in $\mathcal{O}$ and (4) by definition of substitution. The last statement is indeed provable:

$$\dfrac{\dfrac{}{\phi[\sigma_{[x:=e]}]^L, \phi[\sigma_{[x:=e]}]^R}\ \textsc{Hyp}}{\phi[\sigma_{[x:=e]}]^L, (\bigvee x.\ \phi[\sigma'_{[x:=x]}])^R}\ \textsc{RightExists}$$

Secondly, we need to show the *least* upper bound property:

$$\forall a \in O.(\forall e \in O.\, [\![\phi]\!]_{\sigma[x:=e]} \sqsubseteq a) \implies ([(\bigvee x.\ \phi)[\sigma']]_{\dashv\vdash} \sqsubseteq a)$$

which is equivalent to

$$\forall \psi \in \mathcal{T}_{QOL}(X).(\forall e \in O.\, [\![\phi]\!]_{\sigma[x:=e]} \sqsubseteq [\psi]_{\dashv\vdash}) \implies ([(\bigvee x.\ \phi)[\sigma']]_{\dashv\vdash} \sqsubseteq [\psi]_{\dashv\vdash})$$

Fix an arbitrary ψ and assume $\forall e \in O.\, [\![\phi]\!]_{\sigma[x:=e]} \sqsubseteq [\psi]_{\dashv\vdash}$. Consider a variable x_2 that does not appear in ψ. Then, we have in particular,

$$[\![\phi]\!]_{\sigma[x:=x_2]} \sqsubseteq [\psi]_{\dashv\vdash}.$$

Then,

$$\begin{aligned} [\![\phi]\!]_{\sigma[x:=x_2]} \sqsubseteq [\psi]_{\dashv\vdash} &\iff \\ [\phi[\sigma'_{[x:=x_2]}]]_{\dashv\vdash} \sqsubseteq [\psi]_{\dashv\vdash} &\iff \\ \phi[\sigma'_{[x:=x_2]}] \vdash_{\mathbf{QLO}} \psi \text{ is provable} &\iff \end{aligned}$$

Then, using a proof of the last line, we can construct:

$$\frac{\phi[\sigma'_{[x:=x_2]}]^L, \psi^R}{(\bigvee x.\ \phi[\sigma'_{[x:=x]}])^L, \psi^R}\ \textsc{LeftExists}$$

We finally obtain our second property as desired:

$$[(\bigvee x.\ \phi)[\sigma']]_{\dashv\vdash} \ \sqsubseteq\ [\psi]_{\dashv\vdash}$$

To conclude the proof of Lemma 2.7.6, the case with $\bigwedge$ instead of $\bigvee$ is symmetric. □

Hence, our interpretation in $\mathcal{O}$ is guaranteed to be well-defined. However, $\mathcal{O}$ is not guaranteed to be complete for arbitrary sets of elements, which our definition of a model requires. To obtain a complete ortholattice, we will apply MacNeille completion to $\mathcal{O}$.

Definition 2.7.7 (MacNeille completion, [59])**.** Given a lattice L, there exists a smallest complete lattice L' containing L as a sublattice with an embedding $i : L \to L'$ preserving the least upper bounds and greatest lower bounds of arbitrary (possibly infinite) subsets of L. This is the MacNeille completion of L.

Hence, there exists a complete lattice $\mathcal{O}'$ containing $\mathcal{O}$ as a sublattice and preserving the existing least upper bounds and greatest lower bounds. But we also need $\mathcal{O}'$ to be an ortholattice, containing $\mathcal{O}$ as a subortholattice. Fortunately, this is true thanks to a theorem of Bruns.

Lemma 2.7.8 (Theorem 4.2 of [14])**.** For every ortholattice $\mathcal{O}$, its MacNeille completion $\mathcal{O}'$ admits an orthocomplementation which extends the orthocomplementation of $\mathcal{O}$.

Corollary 2.7.9. There exists an injective ortholattice homomorphism $i : \mathcal{O} \to \mathcal{O}'$ such that

$$\forall a, b \in O. a \leq_{\mathcal{O}} b \iff i(a) \leq_{\mathcal{O}'} i(b)$$

and for any $S \subset O$ such that $\bigsqcup S$ (resp. $\bigsqcap S$) exists:

$$\begin{aligned} i(\bigsqcup S) &= \bigsqcup(\{i(x) \mid x \in S\}) \\ i(\bigsqcap S) &= \bigsqcap(\{i(x) \mid x \in S\}). \end{aligned}$$

We can now finish our completeness proof. Define $\sigma : X \to O$ by $\sigma(x) = [x]_{\dashv\vdash}$. Then by Lemma 2.7.6, $[\![\phi]\!]_\sigma = [\phi]_{\dashv\vdash}$. Let γ, δ be the two formulas such that $[\![s]\!]_\sigma \iff$

($[\![\gamma]\!]_\sigma \sqsubseteq [\![\delta]\!]_\sigma$), according to Definition 2.7.3. Recall that the sequent s is not provable by assumption, i.e. $[\![\gamma]\!]_\sigma \not\sqsubseteq [\![\delta]\!]_\sigma$, and hence:

$$[\gamma]_{\dashv\vdash} \not\sqsubseteq_{\mathcal{O}} [\delta]_{\dashv\vdash}$$

from which we deduce

$$i([\gamma]_{\dashv\vdash}) \not\sqsubseteq_{\mathcal{O}'} i([\delta]_{\dashv\vdash})$$

in the ortholattice $\mathcal{O}'$. We now define $\tau : X \to O'$ such that $\tau(x) = i(\sigma(x))$, implying (by induction and Corollary 2.7.9) that for any ϕ,

$$i([\phi]_{\dashv\vdash}) = [\![\phi]\!]_\tau$$

and therefore, in $\mathcal{O}'$:

$$[\![\gamma]\!]_\tau \not\sqsubseteq [\![\delta]\!]_\tau$$

We have hence built a model with the complete ortholattice $\mathcal{O}'$ and the assignment τ in which $[\![\gamma]\!]_\tau \not\sqsubseteq [\![\delta]\!]_\tau$, so $[\![s]\!]_\tau$ does not hold, as desired.

□

The following corollary is a useful application of Lemma 2.7.4 and Lemma 2.7.5.

Corollary 2.7.10. QLO admits the following proof step:

$$\frac{\Gamma[x := \phi], \Delta[x := \phi] \qquad \phi^L, \psi^R \qquad \phi^R, \psi^L}{\Gamma[x := \psi], \Delta[x := \psi]} \textsc{Subst}$$

Proof. If ϕ^L, ψ^R and ϕ^R, ψ^L are provable, then in any complete ortholattice and for any σ, $[\![\phi]\!]_\sigma = [\![\psi]\!]_\sigma$ and hence

$$[\![\Gamma[x := \phi], \Delta[x := \phi]]\!]_\sigma = [\![\Gamma[x := \psi], \Delta[x := \psi]]\!]_\sigma .$$

Hence, by completeness, there always exists a proof of $\Gamma[x := \psi], \Delta[x := \psi]$. □

It is an open question whether *QOL* is decidable or not, and if it is, what is its complexity. *QBF* can simply be decided by performing quantifier elimination, that is, by transforming quantified formulas into equivalent propositional formulas. This is not possible in *QOL*, as we now show.

Definition 2.7.11 (Quantifier elimination)**.** A quantified propositional logic admits *quantifier elimination* if for any formula Q there exists a quantifier-free formula E such that Q and E are equivalent.

Example 2.7.12. QBF, the theory of quantified classical propositional logic, admits quantifier elimination that replaces the quantified proposition $\exists x.F$ with the proposition $F[x := \bot] \vee F[x := \top]$. This quantifier elimination approach is sound over Boolean algebras in general, because a formula is true in $\{\bot, \top\}$ if and only if it is true in any Boolean algebra.

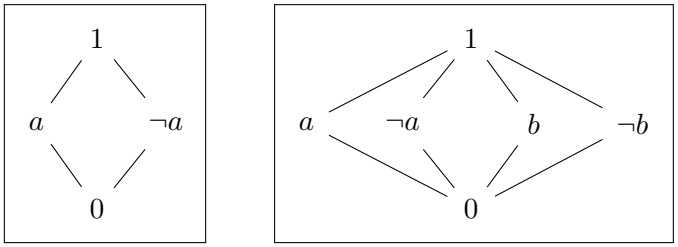


Figure 2.4: The ortholattices M_2 and M_4. M_2 is distributive, but M_4 is not.

Example 2.7.13. The theory of quantified intuitionistic propositional logic does not admit quantifier elimination. Whereas provability in quantifier-free intuitionistic propositional logic corresponds closely to inhabitation in simply typed λ-calculus and is PSPACE-complete [96], the quantified theory corresponds to System F, and is undecidable [30].

In contrast, the next theorem will show that *QOL* does not admit quantifier elimination in general.

Theorem 2.7.14. *QOL* does not admit quantifier elimination. In particular, there exists no quantifier-free formula E such that

$$E \dashv\vdash_{\mathbf{QLO}} \bigvee x.\ \Big(\neg x \wedge (y \vee x)\Big)$$

Proof. For the sake of contradiction, suppose such an E exists. Let $y, w_1, \ldots, w_n$ be the free variables appearing in E. Since $\bigvee x.\ \neg x \wedge (y \vee x)$ is constant with respect to $w_1, \ldots, w_n$, E must be as well, and hence we can assume E only uses y as a variable. Then, the laws of orthologic imply that any quantifier-free formula whose only variable is y is equivalent to one of $0, 1, y$ or $\neg y$. This can easily be shown by induction on the structure of the formula:

$$\begin{array}{llll} 0 \wedge 0 & = 0 & 0 \wedge 1 & = 0 \\ 0 \wedge y & = 0 & 0 \wedge \neg y & = 0 \\ 1 \wedge 1 & = 1 & 1 \wedge y & = y \\ 1 \wedge \neg y & = \neg y & y \wedge y & = y \\ y \wedge \neg y & = 0 & \neg y \wedge \neg y & = \neg y \\ \neg 0 & = 1 & \neg 1 & = 0 \\ \neg\neg y & = y & & \end{array}$$

and similarly for disjunction.
Now, consider the ortholattices M_2 and M_4 in Figure 2.4:
Using soundness of orthologic over ortholattices (Theorem 2.2.5), the formula E (if it exists) needs to be equal to $\bigvee x.\ \neg x \wedge (y \vee x)$ in all models. Since the model is finite, it is straightforward to compute in the ortholattice M_2 with the assignment $y := a$ that:

$$\Big[\!\Big[\bigvee x.\ \neg x \wedge (y \vee x)\Big]\!\Big]_{M_2, y:=a} = a$$

And hence the only compatible formula for E is the atom y.

However, in M_4:

$$\Big[\!\Big[\bigvee x.\ \neg x \wedge (y \vee x) \Big]\!\Big]_{M_4, y:=a} = 1$$

Hence, any expression for E among $0, 1, y, \neg y$ will fail to satisfy at least one of the two examples, and we conclude that there is no quantifier-free formula E that is equivalent to $\bigvee x.\ \neg x \wedge (y \vee x)$. □

This result implies that we can use quantifiers to define new operators, such as

$$\lceil y \rceil \ \equiv\ \bigvee x.\ \Big(\neg x \wedge (y \vee x)\Big)$$

since Theorem 2.7.14 shows that $\lceil y \rceil$ is not expressible without quantifiers.

2.8 Effectively propositional orthologic

In this section, we introduce decidable classes of *predicate* orthologic. This extension is inspired by the Bernays–Schönfinkel–Ramsey (BSR) class of classical first-order logic formulas [11], which consists of formulas of first-order logic that contain predicates and term variables but no function symbols, and whose prenex normal form is of the form $\exists x_1, \ldots, x_m.\ \forall y_1, \ldots, y_n.\ \phi$ where ϕ is quantifier free. Syntactically, a formula in the BSR class can be represented as a quantifier-free formula with constant and variable symbols, where the variables are implicitly universally quantified. It is also called *Effectively Propositional Logic* (EPR) [77], because deciding the validity of such formulas can be reduced to deciding the validity of a formula in propositional logic by a grounding process. This is possible because formulas in the BSR class have finite Herbrand universe [81, p.1798]. BSR class (with its multi-sorted logic generalization) has applications in verification [72].

Thanks to the polynomial decision procedure for propositional orthologic from Section 2.2, we will obtain a decision procedure for effectively propositional orthologic that runs in exponential time in the worst case, whereas known decision procedures for classical EPR are doubly exponential (as the problem falls in the co-NEXPTIME class). Moreover, we show that it becomes polynomial if we restrict the maximal number of variables in axioms, which is in contrast to the corresponding restriction yielding an NP-hard class for classical EPR logic [11].

In this section, *variables* denote variable symbols at the level of terms, and not propositional variables. For generalization, we will formulate the problem over OL^+, including function symbols[1] rather than OL.

Definition 2.8.1. Let C and V be two disjoint countably infinite sets of symbols. Elements of C are called *constants* and elements of V are called *variables*.

A predicate signature Σ is a pair $(P, \bar{C})$ where $P = \{p_1, \ldots, p_n\}$ is a finite set of predicate symbols with their non-negative arities s_i, and $\bar{C} = \{c_1, \ldots, c_m\} \subset C$ is a finite non-empty set of constants.

Define the set of atomic formulas over Σ as

$$Q_\Sigma = \bigcup_{i=1}^{n} \{p_i(\vec{x}) \mid \vec{x} \in (V \cup \bar{C})^{s_i}\}$$

An EPR formula (over Σ) is a formula constructed from Q_Σ using $\wedge, \vee, \neg$ and function symbols, that is, an element of $\mathcal{T}_{OL^+}(Q_\Sigma)$.

Definition 2.8.2. An annotated EPR formula is a^L or a^R, where a is an EPR formula. An EPR sequent is a set of at most two annotated EPR formulas. The degree of a formula or sequent is the number of distinct variables in it. The degree of a finite set A of sequents, $d(A)$, is the maximum of degrees of its sequents. An atomic formula,

[1] At the level of formulas, not terms.

formula, or a sequent is *ground* if it contains no variables (only constants), that is, it has degree zero.

Definition 2.8.3. An EPR deduction instance is a pair (A, S) where A (the axioms) is a set of EPR sequents and S (the goal) is an EPR sequent.

Definition 2.8.4. An instance of a formula (respectively, sequent) is a formula (or sequent) obtained by replacing all occurrences of some variables by any variable or constant. The expansion of a formula s (respectively, sequent), denoted s^*, is the set of all of its instances. s^* is countable, and infinite if s contains at least one variable. If A is a set of sequents then we define $A^* := \bigcup_{s \in A} s^*$. For an EPR deduction instance (A, S) its expansion is (A^*, S).

Definition 2.8.5. EPR-OL-D is the following problem: given a signature Σ and EPR deduction instance (A, S) over Σ, decide whether its expansion (A^*, S) is a valid OL^+ deduction instance (that is, S is a provable consequence of A^* in $\mathbf{LO}^+$).

We first show, as an intermediate lemma, that we only need to look at instances of A using variables appearing in S and constants in S and A.

Lemma 2.8.6. Suppose S has a proof $\mathcal{P}$ involving axioms in A^*. Then it has a proof containing only variables that appear in S and constants in S and A.

Proof. Consider a variable z that appears somewhere in $\mathcal{P}$ but not in S, then it has to be eliminated at some point, and only the Cut rule can remove formulas. Let $c \in \bar{C}$ be any constant symbol of Σ. Let $\mathcal{P}[z := c]$ be the proof $\mathcal{P}$ with every instance of z replaced by c. For any given axiom $a \in A^*$, $a[z := c]$ is also an axiom of A^*, so that all axiom steps occurring in $\mathcal{P}[z := c]$ are correct. It is easy to see that all non-axioms steps in $\mathcal{P}[z := c]$ remain correct under the substitution.

To eliminate constants, we use the same argument except that since axioms are not stable under renaming of constants (but all other rules and in particular the hypothesis rule are), we cannot eliminate constant symbols appearing in A. □

Theorem 2.8.7. The EPR-OL-D problem (A, S) of size n and degree $d(A)$ is in $PTIME(n^{d(A)})$.

Proof. We will reduce EPR-OL-D to the propositional OL^+ deduction problems, which by Theorem 2.2.12 can then be solved in polynomial time. Using completeness of OL^+ (Theorem 2.2.6), the question becomes: can we compute a finite subset A' of A^* such that S has an orthologic proof with axioms among A'?

Lemma 2.8.6 implies that for any sequent S, we only need to consider a finite number of axioms, namely the axioms involving variables in S and constants in A and S (at least one). Each axiom a has at most $(|S| + ||A||)^{d(A)}$ such instances, so the total number of axioms we need to consider is $\mathcal{O}(|A| \cdot (|S| + ||A||)^{d(A)}) = \mathcal{O}(n^{d(A)+1})$. Combining this result with Theorem 2.2.12 gives $PTIME(n^{d(A)})$, or, more precisely, $\mathcal{O}(n^{3(d(A)+1)})$. □

2.8.1 Instantiation as a rule

Instead of starting by grounding all axioms, we can delay instantiation until later in the proof, yielding shorter proofs in some cases. Formally, we add an instantiation step to the proof calculus of $\mathbf{LO}^+$ (Definition 2.2.3) over $\mathcal{T}_{OL^+}(Q_\Sigma)$:

Definition 2.8.8. Let $\mathbf{LO}^{+\boldsymbol{I}}$ be the proof system obtained by adding to the proof system $\mathbf{LO}^+$ over $\mathcal{T}_{OL^+}(Q_\Sigma)$ the following rule:

$$\frac{\Gamma, \Delta}{\Gamma[\vec{x} := \vec{t}], \Delta[\vec{x} := \vec{t}]}\ \textsc{Inst}$$

holding for arbitrary sets of term variables $\vec{x}$ and terms $\vec{t}$.

Lemma 2.8.9. For a sequent S and set of axioms A over $\mathcal{T}_{OL^+}(Q_\Sigma)$, S has an $\mathbf{LO}^+$ proof from A^* if and only if S has an $\mathbf{LO}^{+\boldsymbol{I}}$ proof from A.

Proof.

$\rightarrow$: Given an $\mathbf{LO}^+$ proof with axioms in A^*, said axioms can be obtained from A in $\mathbf{LO}^{+\boldsymbol{I}}$ by an application of the instantiation rule:

$$\dfrac{\dfrac{}{\Gamma^*, \Delta^*}\ \textsc{Axiom}}{\vdots\ S} \quad \hookrightarrow \quad \dfrac{\dfrac{\dfrac{}{\Gamma, \Delta}\ \textsc{Axiom}}{\Gamma^*, \Delta^*}\ \textsc{Inst}}{\vdots\ S}$$

where $(\Gamma, \Delta) \in A$ and $(\Gamma^*, \Delta^*) \in A^*$.

$\leftarrow$: Note that if the Inst rule is only used right after axioms, then we can reverse the transformation above. We show that given a proof in $\mathbf{LO}^{+\boldsymbol{I}}$ using Inst, the instances of Inst can be swapped with other rules and be pushed to axioms. For example

$$\dfrac{\dfrac{\dfrac{\mathcal{A}'}{\alpha^L, \Delta} \quad \dfrac{\mathcal{A}''}{\beta^L, \Delta}}{(\alpha \vee \beta)^L, \Delta}\ \textsc{LeftOr}}{(\alpha^* \vee \beta^*)^L, \Delta^*}\ \textsc{Inst} \quad \hookrightarrow \quad \dfrac{\dfrac{\dfrac{\mathcal{A}'}{\alpha^L, \Delta}}{(\alpha^*)^L, \Delta^*}\ \textsc{Inst} \quad \dfrac{\dfrac{\mathcal{A}''}{\beta^L, \Delta}}{(\beta^*)^L, \Delta^*}\ \textsc{Inst}}{(\alpha^* \vee \beta^*)^L, \Delta^*}\ \textsc{LeftOr}$$

The cases for all other rules are similar. Then, any conclusion of an Inst rule is a member of A^* and can be replaced by an Axiom rule to obtain an $\mathbf{LO}^+$ proof from A^*.

□

$$\dfrac{\dfrac{\Gamma, \phi^R}{\theta(\Gamma), \theta(\phi)^R}\,\textsc{Inst} \qquad \dfrac{\psi^L, \Delta}{\theta(\psi)^L, \theta(\Delta)}\,\textsc{Inst}}{\theta(\Gamma), \theta(\Delta)}\,\textsc{Cut}$$

where θ is the most general unifier of ϕ and ψ

$$\dfrac{\dfrac{\Gamma_1, \phi^R}{\theta(\Gamma_1), \theta(\phi)^R}\,\textsc{Inst} \qquad \dfrac{\Gamma_2, \psi^R}{\theta(\Gamma_2), \theta(\psi)^R}\,\textsc{Inst}}{\theta(\Gamma_1), \theta(\phi \wedge \psi)^R}\,\textsc{RightAnd}$$

$$\dfrac{\dfrac{\Gamma_1, \phi^L}{\theta(\Gamma_1), \theta(\phi)^L}\,\textsc{Inst} \qquad \dfrac{\Gamma_2, \psi^L}{\theta(\Gamma_2), \theta(\psi)^L}\,\textsc{Inst}}{\theta(\Gamma_1), \theta(\phi \vee \psi)^L}\,\textsc{LeftOr}$$

where θ is the most general unifier of Γ_1 and Γ_2

Figure 2.5: Sequent-calculus style deduction rules with unification for Effectively Propositional Orthologic.

2.8.2 Proof search with unification

While searching for a proof, we usually want to delay decision-making (such as which variable to instantiate) for as long as possible. In backward proof search, this means we want to delay it until the sequent is an axiom of A^*. In forward proof search, however, being able to use the Inst rule allows delaying instantiation as much as possible, as in resolution for classical first-order logic [81, Chapter 2].

We thus adopt unification to decide when and how to instantiate a variable, whenever we use a rule with two premises. The corresponding directed rules are shown in Figure 2.5. Since EPR only admits constant symbols and no term-level function symbols, the most general unifier of ϕ and ψ is the substitution θ of smallest support such that $\theta(\phi) = \theta(\psi)$ [81, Chapter 8].

Theorem 2.8.10. A sequent S over $\mathcal{T}_{OL^+}(Q_\Sigma)$ has a proof in $\mathbf{LO}^{+\boldsymbol{I}}$ if and only if there exists a sequent S' such that S is a particular instance of S' and S' has a proof where the Inst rule is only used through unification (as in the specific cases of Figure 2.5) or to rename variables.

Proof. (Sketch). The proof is once again by induction and case analysis on the proof $\mathcal{P}$ of S, except we move the instantiation step toward the conclusion of the proof. Consider the proof rule that follows the instantiation:

- Hyp is a leaf rule, so it can never follow a step.
- Replace is immediate, as long as the variables in Δ are properly renamed.

$$\dfrac{\dfrac{\dfrac{\mathcal{A}}{\Gamma,\Gamma}}{\sigma(\Gamma),\sigma(\Gamma)}\textsc{Inst}}{\sigma(\Gamma),\Delta}\textsc{Replace} \quad\hookrightarrow\quad \dfrac{\dfrac{\dfrac{\mathcal{A}}{\Gamma,\Gamma}}{\Gamma,\pi(\Delta)}\textsc{Replace}}{\sigma(\Gamma),\Delta}\textsc{Inst}$$

where π is a renaming of variables in Δ to names that are fresh. In particular, it is invertible.

- LeftNot and RightNot are immediate.
- For LeftAnd and RightOr, the transformation is the same as for Replace.
- In theCut case, assume that the two premises (Γ, ϕ^R) and (ψ^L, Δ) have no shared variables, by renaming them if necessary:

$$\dfrac{\dfrac{\Gamma,\phi^R}{\sigma_1(\Gamma),\sigma_1(\phi)^R}\textsc{Inst} \quad \dfrac{\psi^L,\Delta}{\sigma_2(\psi)^L,\sigma_2(\Delta)}\textsc{Inst}}{\sigma_1(\Gamma),\sigma_2(\Delta)}\textsc{Cut}$$

$$\hookrightarrow$$

$$\dfrac{\dfrac{\dfrac{\Gamma,\phi^R}{\theta(\Gamma),\theta(\phi)^R}\textsc{Inst} \quad \dfrac{\psi^L,\Delta}{\theta(\psi)^L,\theta(\Delta)}\textsc{Inst}}{\theta(\Gamma),\theta(\Delta)}\textsc{Cut}}{\sigma_1(\Gamma),\sigma_2(\Delta)}\textsc{Inst}$$

Note that since θ is the most general unifier for ϕ and ψ, and $\sigma_1(\phi) = \sigma_2(\psi)$, θ factors through both σ_1 and σ_2, so that the last Inst step is correct.

- LeftOr and RightAnd are similar to theCut step.

□

In practice, in a forward proof search, formally renaming variables for each sequent is not needed. We can simply always consider variables in different sequents to be distinct.

2.8.3 Solving and extending Datalog programs with orthologic

Datalog is a logical and declarative programming language admitting formulas in a further restriction of the BSR class where ϕ is forced to be a Horn clause (over predicates). A Datalog program is then a conjunction of such formulas (or clauses) [94, 95, 24]. While the validity problem for the BSR class is co-NEXPTIME-complete [80, 77], solving a Datalog program is only EXPTIME-complete. This makes Datalog a suitable language for logic programming and database queries. Typically, a Datalog query asks if a certain fact (an atom without variables) is a consequence of the clauses in the program. This naturally corresponds to solving a deduction problem, with the axioms corresponding to the program and the goal to the query.

Definition 2.8.11. A *Datalog program* is an EPR deduction instance where all axioms are Horn sequents, i.e., of the form $(a_1 \wedge \ldots \wedge a_n)^L, b^R$ where $a_i, b \in Q_\Sigma$ are atomic formulas.

Note that when interpreted over classical logic, this corresponds to actual Datalog programs.

Lemma 2.8.12. Datalog programs are valid in classical predicate logic if and only if they are valid in predicate orthologic.

Proof. Theorem 2.8.7 shows that EPR problems can be reduced via grounding to propositional OL^+. Moreover, as the resulting set of axioms contains only Horn clauses, Corollary 2.6.6 implies that the corresponding deduction problem has the same semantics in orthologic and classical logic. □

Corollary 2.8.13. Datalog programs can be evaluated using orthologic in time $PTIME(n^{d(A)})$.

Proof. Combine Lemma 2.8.12 with Theorem 2.8.7. □

This matches known complexity classes of Datalog, which has exponential query complexity (corresponding here to axioms with variables) and polynomial data complexity (corresponding to the goal and variable-free axioms, or *facts*) [24].

2.8.4 Axiomatizing congruence and equality relations

We now show that, when equality is axiomatized as a congruence relation in effectively propositional orthologic, a substitution rule becomes admissible. Let us denote elements of V (variables) by $x, y, z, \ldots$. Let $\Sigma_\sim$ be a signature with arbitrary predicate symbols $p_1, \ldots, p_n$ each of arity s_i and one distinguished predicate symbol $\sim$ representing a congruence relation. Consider the set of axioms $A_\sim$ containing the following sequents that axiomatize the equivalence property:

$$\begin{aligned} (x \sim x)^R \\ (x \sim y)^L, (y \sim x)^R \\ (x \sim y \wedge y \sim z)^L, (x \sim z)^R \end{aligned}$$

and for each symbol p_i and each $1 \leq j \leq s_i$, the congruence property for p_i:

$$(x \sim y \wedge p_i(z_1, \ldots, z_{j-1}, x, z_{j+1}, \ldots, z_{s_i}))^L, p_i(z_1, \ldots, z_{j-1}, y, z_{j+1}, \ldots, z_{s_i})^R$$

Again, this does not constitute a finite presentation of an ortholattice, as there are infinitely many possible instances of axioms. However, by Theorem 2.8.7, if a sequent S over $\Sigma_\sim$ has a proof involving axioms in $A_\sim$, then it has a proof with only variables that appear in S. Moreover, the degree of $A_\sim$ is $d(A_\sim) = \max(3, \max_i(s_i) + 1)$, so that the complexity of the proof search is exponential in the arity of the predicates in the

language and polynomial in the size of the problem, for a fixed language. The following lemma shows that in any decision problem whose axioms contain $A_\sim$, we can add a substitution rule for equality.

Lemma 2.8.14. Let A be a set of axioms such that $A_\sim \subset A$. The following rule for substitution of equal terms is admissible in $\mathbf{LO}^{+\boldsymbol{I}}$ with axioms in A:

$$\frac{\Gamma[x := s], \Delta[x := s] \qquad (s \sim t)^R}{\Gamma[x := t], \Delta[x := t]}\ \textsc{Subst}_\sim$$

Proof. Suppose x occurs only once in Γ, Δ (if it appears multiple times, we repeat the argument). Suppose without loss of generality that this unique occurrence is in Γ. Let $a(x) \equiv p_i(u_1, \ldots, x, \ldots, u_{s_i})$ be the atomic formula containing this occurrence of x, i.e. $\Gamma = \Gamma'[\chi := a(x)]$, for a propositional variable χ. Axioms in $A_\sim$ allow the following proof, where we first derive $a(s)^L, a(t)^R$:

$$\dfrac{\dfrac{\dfrac{\dfrac{s \sim t^R}{a(s)^L, s \sim t^R}\ \textsc{Replace} \quad \dfrac{}{a(s)^L, a(s)^R}\ \textsc{Hyp}}{a(s)^L, (s \sim t \wedge a(s))^R}\ \textsc{RightAnd} \quad \dfrac{}{(s \sim t \wedge a(s))^L, a(t)^R}\ \textsc{Axiom}}{a(s)^L, a(t)^R}\ \textsc{Cut}$$

and then conclude, using an analogous derivation of $a(t)^L, a(s)^R$:

$$\dfrac{\dfrac{\Gamma[x := s], \Delta}{\Gamma'[\chi := a(s)], \Delta} \qquad a(s)^L, a(t)^R \qquad \dfrac{\dfrac{\dfrac{(s \sim t)^R \qquad \dfrac{}{(s \sim t)^L, (t \sim s)^R}\ \textsc{Axiom}}{(t \sim s)^R}\ \textsc{Cut}}{\text{...(analogous)...}}}{a(t)^L, a(s)^R}}{\dfrac{\Gamma'[\chi := a(t)], \Delta}{\Gamma[x := t], \Delta}}\ \textsc{Subst}$$

where Subst is the substitution rule that we showed admissible in Corollary 2.2.7 and the dashed lines denote syntactic equality. Hence, Subst $\sim$ is admissible in orthologic with any axiomatization containing $A_\sim$. □

2.9 Orthologic-based type system

The previous sections studied the proof theory of orthologic, that is, properties of expressions in the language of orthologic. There is a particular domain where deciding inequality between such expressions arises naturally: *subtyping*.

In programming languages, subtyping is a relation between types, allowing one to be considered a special case of another. Concretely, if A is a subtype of B (denoted $A <: B$), then any expression of type A also has type B. Classical examples are that the class of cats is a subtype of the class of animals, since any cat is also an animal, or that the type of both integers and floats are subtypes of the type of numbers.

Listing 2.11: Example of subtyping relations in Scala

```
class Animal:
class Cat extends Animal:
  def speak(): String = "Meow"
val a: Animal = new Cat()

def x: Number = (3: Int)
def y: Number = (3.4: Float)
```

Types form a partial order (Definition 2.1.16) under the subtyping relation, meaning that it is reflexive, transitive and antisymmetric. Some languages offer *union* and *intersection* of any two types. Union types are denoted $A|B$ and intersection types $A\&B$ in Scala, so we will adopt this notation as well. Union and intersection types satisfy the properties

$$A <: A|B$$

$$B <: A|B$$

$$A <: C \text{ and } B <: C \implies A|B <: C$$

and dually for &, making the space of types a lattice (Definition 2.1.15). If both $A <: B$ and $B <: A$, we write $A =:= B$ and say that A and B are equivalent types.

Union and intersection types have many uses. For example, union types can be used to define a supertype of some types after they are already defined. Indeed, consider the following example:

Listing 2.12: Example of union types in Scala

```
sealed trait Human
class Kid extends Human
class Teenager extends Human
class Adult extends Human
```

Declaring new types or classes as subtypes of existing classes is called *nominal subtyping*. Here, the type `Human` is essentially equivalent to the union type `Kid | Teenager | Adult`. However, nominal subtyping does not allow defining new supertypes, so that there is no way, a posteriori, to define a type `Minor` that would

contain only the members of `Kid` and `Teenager`. This can be achieved with union types.

```
1 type Minor = Kid | Teenager
```

However, supporting union and intersection types introduces algorithmic challenges, and industrial programming languages supporting them (such as Scala, TypeScript or Flow as of 2025) exhibit unsatisfactory behaviours in such cases, with either inefficient type checking algorithms, unintuitive typing rules, incompleteness or even unsound behaviours. The proof search algorithm for orthologic suggests an alternative approach: $A <: B$ holds if and only if it is provable from the laws of (ortho)lattices.

This is not an obvious or the only approach, as in principle nothing prevents type systems from satisfying more laws than those of ortholattices. This means that we consider the set of types as being the *free (ortho)lattice*. Additionally, this yields support for *negation types* $\neg A$, which can be interpreted as the type of things that are not of type A. If such types are not syntactically supported, then deciding an inequality of types with only union and intersection types over ortholattices is equivalent to deciding it over lattices.

A type system with only base types and union, intersection and negation types would be quite limited. In practice, much of the expressiveness of type systems comes from *type constructors*, which are type-level functions taking types as arguments, and returning a new type. For example, the type constructor `List` takes a type A and returns the type `List[A]`, the type of lists of elements of type A. Type constructors can be covariant (if $A <: B$ then $List[A] <: List[B]$), contravariant (meaning, if $A <: B$ then $F[B] <: F[A]$), invariant (no subtyping relation is induced). Type constructors can also have multiple parameters, each of different variance, such as the type $A \Rightarrow B$ of functions taking an argument of type A and returning a value of type B. $\Rightarrow$ is contravariant in its first argument and covariant in the second: if $B <: A$ and $C <: D$ then $A \Rightarrow C <: B \Rightarrow D$. Representing type constructors is in fact the main motivation behind studying OL^+.

2.9.1 Simple language

To concretely describe how orthologic-based subtyping applies to type checking, we define a small language with expressions and types in the spirit of Hindley–Milner's algorithm (for System Fω with polymorphism restricted to the top level), with pattern matching and with covariant and contravariant type constructors.

Definition 2.9.1 (Types)**.** Let $\mathcal{B}$ be an arbitrary fixed set of base type constructors, containing in particular arity-0 constructors such as `Bool`, `Int`, `String`, etc. Let $\mathcal{V}$ be a

countably infinite set of type variables. Let *pre-types* be given by the grammar

$$\begin{aligned}\mathcal{T} := \ & \mathcal{B}[\mathcal{T}, ...] \\ & \mid \mathcal{V} \\ & \mid \mathcal{T} \to \mathcal{T} \\ & \mid \mathcal{T} \vee \mathcal{T} \\ & \mid \mathcal{T} \wedge \mathcal{T} \\ & \mid \neg\mathcal{T} \\ & \mid \top \\ & \mid \bot\end{aligned}$$

We define *simple types* as the types which do not contain any type variables. As usual, if $T \in \mathcal{T}$ is a type, we write $T(V_1, \ldots, V_n)$ to indicate that the variables of T are among $\{V_1, \ldots, V_n\}$, and $\mathcal{T}(V_1, \ldots, V_n)$ for the set of expressions with variables among $\{V_1, \ldots, V_n\}$.

Then, let *clauses* be defined by the grammar

$$\begin{aligned}\mathcal{C} := \ & \epsilon \\ & \mid \mathcal{T} <: \mathcal{T},\ \mathcal{C}\end{aligned}$$

Finally, we define *polymorphic types* as

$$\mathcal{H} := \forall V_1, \ldots, V_n \ \texttt{where}\ C_1, \ldots, C_m.\mathcal{T}(V_1, \ldots, V_n)$$

In type theory, inhabitants of types are typically called terms. However, this conflicts with our universal algebra terminology, where elements of ortholattices (so, types) are also terms. To avoid the confusion, we will use *expressions* to speak of (generalized) λ-expressions inhabiting types.

Definition 2.9.2 (Expressions)**.** Let $\mathbb{L}$ be a set of literals containing, for example, (`1, 2, true, false, +,`...) and let $\mathbb{V} := \{x_i\}$ be an arbitrary set of variable symbols. The set of *expressions* is defined as

$$\begin{aligned}\mathbb{E} := & \mathbb{L} \\ & \mid \mathbb{V}[\mathcal{T}, ...] \\ & \mid \mathbb{E}\ \mathbb{E} \\ & \mid \lambda\mathbb{V} : \mathcal{T}.\mathbb{E} \\ & \mid \ \texttt{def}\ \mathbb{V}[\mathcal{V}_1, \ldots, \mathcal{V}_n \ \texttt{where}\ \mathcal{C}_1, \ldots, \mathcal{C}_m] : \mathcal{T}\ \texttt{:=}\ \mathbb{E}\ \texttt{in}\ \mathbb{E} \\ & \mid \ \texttt{match}\ \mathbb{E}\ \texttt{with}\ \mathbb{P}\end{aligned}$$

where

$$\mathbb{P} := \epsilon \mid \mathbb{V} : \mathcal{T} \Rightarrow \mathbb{E}, \mathbb{P}$$

are called *patterns*.

Example 2.9.3. The following is an expression

```
def ntimes[A where A <: Semigroup[A]] : A ⇒ Int ⇒ A ⇒ A :=
  λ base: A. λn: Int. λelem: A.
    if_then_else[A] (eq[Int] n 0)
      base
      (combine[A] elem (ntimes[A] base (sub n ) elem))
in
ntimes[Int] 0 3 4
```

Assuming that `combine`, `sub`, `0`, `1`, `3`, `4` $\in \mathbb{L}$ and `Semigroup`, `Int` $\in \mathcal{B}$.

Definition 2.9.4 (Typing judgements)**.** A typing context is a pair (C, Γ) where C is a set of clauses and Γ is a set of pairs $(x : T)$ where $x \in \mathbb{V}$ and $T \in \mathcal{T}$.

A typing judgement is a tuple $((C, \Gamma), e, T)$ where (C, Γ) is a typing context, $e \in \mathbb{E}$ is an expression and $T \in \mathcal{T}$ is a type. We write it as $C; \Gamma \vdash e : T$.

We define inductively the subset of *valid* type judgements according to the derivation rules of Figure 2.6.

Note that this system is algorithmic in the following sense: every expression has at most one type, which the proof system computes.

Theorem 2.9.5. For all Γ and C, for all $e \in \mathbb{E}$, there exists at most one $T \in \mathcal{T}$ such that $C; \Gamma \vdash e : T$. Additionally, T is computed in linear time in the size of e.

Proof. By induction on the structure of e. Note that for each form of expression, at most one typing rule can apply. Moreover note that the resulting type in a typing rule is given by the type of a subexpression e' (which by induction is computable in time proportional to the size of e') and optionally of parameters explicit in e. The algorithm is given in Listing 2.13. □

Constrained polymorphism The **`def`** construct allows defining recursive polymorphic functions, with arguments ranging over types. Most languages with subtyping offer *bounded* polymorphism [26], for example **`def`** `foo[T<:B]`, where `foo` is only polymorphic over subtypes of `B`. Some languages also offer lower bounds. Because our orthologic checker supports reasoning with arbitrary assumptions, we support arbitrary constraints on type variables, for example **`def`** `bar[T` **`where`** `F[T] <: G[T]]`, for arbitrary expressions `F[T]` and `G[T]`.

Inside the body of the function, this amounts to type checking with the additional assumption given by the constraint. At a call site, a subtyping check needs to be

$$\frac{}{C; \Gamma, (x : T) \vdash x : T}\ \text{Var} \qquad \frac{C; \Gamma, (x : T_1) \vdash e : T_2}{C; \Gamma \vdash (\lambda x : T_1 . e) : T_1 \to T_2}\ \text{Lambda}$$

$$\frac{C; \Gamma \vdash e : T_1 \to T_2 \qquad C; \Gamma \vdash e' : T_3 \qquad C \vdash T_3 <: T_1}{C; \Gamma, \vdash (e\, e') : T_2}\ \text{App}$$

$$\frac{C; \Gamma \vdash e : \forall A_1, \ldots, A_n\ \texttt{where}\ C_1, \ldots, C_m . T \qquad \text{for all } i,\ C \vdash C_i[A_1 := T_1, \ldots, A_n := T_n]}{C; \Gamma, \vdash e[T_1, \ldots, T_n] : T[A_1 := T_1, \ldots, A_n := T_n]}\ \text{TypeApp}$$

$$\frac{\begin{array}{c} C, C_1, \ldots, C_m; \Gamma, f : \forall A_1, \ldots, A_n\ \texttt{where}\ C_1, \ldots, C_m . T \vdash e_1 : T_1 \\ C, C_1, \ldots, C_m \vdash e : T_1 <: T \\ C; \Gamma, f : \forall A_1, \ldots, A_n\ \texttt{where}\ C_1, \ldots, C_m . T \vdash e_2 : T_2 \end{array}}{C; \Gamma, \vdash\ \texttt{def}\ f[A_i\ \texttt{where}\ C_j] : T := e_1\ \texttt{in}\ e_2 : T_2}\ \text{LetRec}$$

$$\frac{C; \Gamma, \vdash e : S \qquad C \vdash S <: \bigvee_i T_i \qquad C; \Gamma, x_i : T_i \vdash e_i : T_i'}{C; \Gamma \vdash\ \texttt{match}\ e\ \texttt{with}\ x_i : T_i \Rightarrow e_i : \bigvee_i T_i'}\ \text{Match}$$

Figure 2.6: Typing rules for a simple language with orthologic-based subtyping.

Listing 2.13: Type checking algorithm. The function `OL.checkLEQ` implements the subtyping algorithm of Theorem 2.2.12.

```
def typeCheck(C: List[Clause], Γ: Map[𝕍, 𝒯], e: 𝔼): 𝒯 = e match {
  case Var(x) ⇒ Γ(x)
  case Lam(x, t, e2) ⇒
    val Γ₂ = Γ + (x → t)
    FunType(t, typeCheck(C, Γ₂, e2))
  case App(e, e2) ⇒
    val t = typeCheck(C, Γ, e)
    val t2 = typeCheck(C, Γ, e2)
    t match {
      case FunType(paramType, returnType)
        if OL.checkLEQ(t2, paramType, C) ⇒ returnType
      case _ ⇒ throw new TypeError("Type mismatch")
    }
  case LetRec(f, Ai: List[𝕍], Ci: List[Clause], t: 𝒯, e: 𝔼, e2: 𝔼) ⇒
    val Γ₂ = Γ + (f → PolyType(Ai, Ci, t))
    val t = typeCheck(C ++ Ci, Γ₂, e)
    if (!OL.checkLEQ(t, t, C ++ Ci)) throw new TypeError("Type mismatch")
    else typeCheck(C, Γ₂, e2)
  case Match(scrutinee: 𝔼, cases: List[(𝕍, 𝒯, 𝔼)]) ⇒
    val s = typeCheck(C, Γ, scrutinee)
    if !OL.checkLEQ(s, ⋁(cases.map(_._2)), C) then
      throw new TypeError("Type mismatch")
    val caseTypes = cases.map { case (x, ti, ei) ⇒
      val Γ₂ = Γ + (x → ti)
      typeCheck(C, Γ₂, ei)
    }
    else ⋁(caseTypes)
}
```

performed for the constraint, instantiated with the adequate type. Such *constrained polymorphism* offers a lot of expressibility. Here are some examples:

```
def f[S, T where S & T <: Nothing]  //only accepts pairs of disjoint types
def g[T where T <: Comparable[T]]   //F-bounded polymorphism
def h[T where T <: Comparable[T], Set[T] <: Comparable[Set[T]]]
                                  //Multiple constraints are allowed

class List[A]: //assuming Scala-style object-oriented programming
  def toMap[S, T where A <: (S, T)] //Constrains types from outer scope
```

Pattern matching Our term language also permits matching expression on types. Operationally, this may be computed by dynamic type tests, again using the algorithm of Theorem 2.2.12. This requires maintaining types at runtime, which may be too large an overhead or conflict with other imperatives. Under type erasure, matching can be restricted to concrete classes, as in many languages.

Type simplification Simplifying the representation of types can improve performance and give better messages to the user. All compilers of mainstream typed programming languages perform some kind of type normalization. For example, consider the expression `if e then a else b` (which can be seen as syntactic sugar for the function `ite: ∀ A. Boolean ⇒ A ⇒ A ⇒ A`). It is natural to infer the type `A|B` for this expression, when `a:A` and `b:B`. However, it will often be the case that `A` and `B` are the same type, for example `Int`. In this case it is desirable to simplify `Int|Int` to `Int` before continuing to type check the program.

Similarly, computing a normal form for types (that is, mapping equivalent types to the same representation) simplifies algorithms and allows for various optimizations, such as the use of caching. However, simplification and normalization are not necessarily compatible: In Boolean algebra for example, CNF and DNF are normal forms but they are not simplifications, as they can increase (exponentially) the representation of an expression. Hence, languages such as TypeScript do not compute a full normal form and instead apply practical heuristics.

On the other hand, the OL^+-based subtyping system benefits from the OL^+ normalization procedure of Theorem 2.4.13 that can never increase the size of types. Such normalization can serve as a basis for simplification of types in type checkers.

We do not define an evaluation semantics for our language, as it is orthogonal to the topic of subtyping with orthologic, and except for pattern matching, which we already discussed, evaluation can be defined independently of the type system.

2.9.2 Classes and abstract type definitions

Classes[2], much like records, provide structure to encapsulate data. However, while subtyping between records is entirely structural, subtyping between classes is often *declarative*. Consider the following Scala-like example:

```
1 trait S
2 trait T[A]
3 class U extends S with T[S]
```

which introduces three types S, $T[A]$ and U such that $U <: S$ and $U <: T[S]$. Note that these two conditions are equivalent to $U <: S \wedge T[S]$. This is directly representable in OL^+ by globally adding this constraint to our context C and new constructors and accessors to Γ. Then by soundness and completeness of OL^+, the provable subtyping relations under the new context are exactly those that are true in all ortholattices satisfying all the axioms of C, $U <: S$ and $U <: T[S]$.

The situation is more complex if U is a polymorphic type, as in for example:

```
1 trait S[A]
2 trait T[A]
3 class U[A] extends S[A] with T[S[A]]
```

as the corresponding constraint becomes

$$U(A) <: S(A) \wedge T(S(A))$$

which must be understood as universally quantified over all A. Our system, however, cannot support arbitrary quantified hypotheses, as they would allow axiomatizing distributivity ($\forall A, B, C. A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$), making the underlying ortholattice a Boolean algebra and the entailment problem co-NP complete. In fact, it would allow encoding any arbitrary variety, by introducing fresh (non-variant) function symbols of appropriate arity and stating the appropriate universally quantified axioms.

Luckily, there are important classes of quantified assumptions that we can nonetheless represent efficiently in OL^+, such as *abstract type definitions*

Definition 2.9.6 (Abstract type definition)**.** We extend our term language with *type definitions* which are of the form

$$\texttt{type}\ T[A_1, A_2, ...] <: F_{A_1,A_2,...}\ \ \texttt{in}\ \ \mathbb{E}$$

In a given context C, Γ, T must be a new constant type that does not appear in F, C or Γ, A_i are type variables assumed to be fresh for C and $\mathtt{F}_{A_1,A_2,...}$ is an arbitrary type whose variables are among the A_i. The corresponding typing rule should intuitively be

$$\frac{C \cup \{T[A_1, A_2, ...] <: F_{A_1,A_2,...} \mid A_1, A_2, ... \in \mathcal{T}_{OL^+}(X)\}; \Gamma \vdash e : S}{C; \Gamma \vdash\ \texttt{type}\ T[A_1, A_2, ...] <: F_{A_1,A_2,...}\ \texttt{in}\ e : S}\ \text{TypeDef}$$

[2]Here understood as comprising traits as well as interfaces, abstract classes and other similar user-defined constructs.

The example above corresponds to the three type definitions:

```
type S[A] <: ⊤
type T[A] <: ⊤
type U[A] <: S[A] ∧ T[S[A]]
```

When T is polymorphic, the definition introduces infinitely many axioms, one for each instantiation of the type variables. However, we can show that this is equivalent to a single axiom, by replacing T by a new type G defined as in the following theorem.

Theorem 2.9.7. Given a context C and an abstract type definition $T[A_1, A_2, ...] <: F_{A_1,A_2,...}$, let $C' := C \cup \{T[A_1, A_2, ...] <: F_{A_1,A_2,...} \mid A_1, A_2, ... \in \mathcal{T}_{OL^+}(X)\}$, i.e. C' is C extended with all the (infinitely many) instantiations of the assumption.

Then there exists a type $G[A_1, A_2, ...]$ such that for any two types S_1 and S_2,

$$S_1 <:_{C'} S_2$$

if and only if

$$S_1[T := G] <:_C S_2[T := G]$$

Proof. Define $G[A_1, A_2, ...] := F_{A_1,A_2,...} \wedge T'[A_1, A_2, ...]$. Let

$$C' := \cup \forall A_1, A_2, ...T[A_1, A_2, ...] <: F_{A_1,A_2,...}$$

- ($\Longrightarrow$) Assume $S_1 <:_{C'} S_2$. By Lemma 2.2.8, we have

  $$S_1[T := G] <:_{C'[T:=G]} S_2[T := G]$$

  Note that for any particular instantiation of the A_i's,

  $$G_{A_1,A_2,...} = F_{A_1,A_2,...} \wedge T'[A_1, A_2, ...] <: F_{A_1,A_2,...}$$

  so the axioms are redundant. Hence

  $$S_1[T := G] <:_C S_2[T := G]$$

- ($\Longleftarrow$) Assume $S_1[T := G] <:_C S_2[T := G]$. Since $C \subseteq C'$, we have

  $$S_1[T := G] <:_{C'} S_2[T := G]$$

  By Lemma 2.2.8 and since C' does not contain T', we have that

  $$S_1[T := G][T' := T] <:_{C'} S_2[T := G][T' := T]$$

  Since T' only appears inside G, this is equal to

  $$S_1[T := G[T' := T]] <:_{C'} S_2[T := G[T' := T]]$$

But note that

$$G[T' := T][A_1, A_2, ...] = F_{A_1,A_2,...} \wedge T[A_1, A_2, ...] =_{C'} T[A_1, A_2, ...]$$

using the absorption law. Hence we obtain $S_1 <:_{C'} S_2$.

□

The TypeDef rule is hence equivalent to the following:

$$\frac{C; \Gamma \vdash e[T[A_1, A_2, ...] := (F_{A_1,A_2,...} \wedge T[A_1, A_2, ...])] : S}{C; \Gamma \vdash \ \texttt{type}\ T[A_1, A_2, ...] <: F_{A_1,A_2,...}\ \texttt{in}\ e : S}\ \text{TypeDef}_{<:}$$

Note that the dual construction allows defining types with upper bounds such as

$$\texttt{type}\ T[A_1, A_2, ...] >: F_{A_1,A_2,...}$$

We can finally handle *type aliases* of the form

$$\texttt{type}\ T[A_1, A_2, ...] = F_{A_1,A_2,...}$$

By simply unfolding T, as compilers do in e.g. Scala and Java, e.g.

$$\frac{C; \Gamma \vdash e[T[A_1, A_2, ...] := F_{A_1,A_2,...}] : S}{C; \Gamma \vdash \ \texttt{type}\ T[A_1, A_2, ...] = F_{A_1,A_2,...}\ \texttt{in}\ e : S}\ \text{TypeDef}_{=}$$

Note on complexity Type aliases can be used to give a shorter representation to types. For example, defining

$$\texttt{type}\ C[A] = B[A, A]$$

implies that $C[C[C[A]]] = B[B[B[A, A], B[A, A]], B[B[A, A], B[A, A]]]$. This unfolding can yield exponentially longer types, but they can be efficiently represented using structure sharing. Since our algorithms of Theorem 2.2.12 and Theorem 2.4.13 only care about the number of unique subexpression of the input, this unfolding has no asymptotic consequence on the runtime.

However, there is a second way in which type aliases can expand to long representation. Consider the following definitions:

```
type C₀[A]
type C₁[A] = C₀[C₀[A]]
type C₂[A] = C₁[C₁[A]]
...
type Cₙ[A] = Cₙ₋₁[Cₙ₋₁[A]]
```

in particular, $C_n[A] = C_0^{2^n}[A]$, and so polymorphic type aliases enable an exponential speed-up in representation with respect to the number of definitions (but not with respect to the size of types). The phenomenon is similar with bounded abstract types.

Nominal equirecursive types Equirecursive types are characterized by a recursive relation. As an example, consider the recursively-defined type of JSON data:

```
type JSON = Double | String | Boolean | Null | Map[Str, JSON] | Seq[JSON]
```

Such recursive definitions using untagged unions are typically considered difficult to reason about, in contrast to their versions that use tagged unions. In our OL^+-based approach, it is straightforward to model such definitions without relying on unfolding: they are simple constraints. Under such semantics, the types are nominal in the sense that the above `JSON` would not be considered equal to another type defined using an alpha-equivalent definition. As mentioned earlier, this cannot generalize to polymorphic recursive definitions, since it would allow to encode all of equational logic, which is undecidable. Hence, we can extend our language with recursive non-polymorphic type definitions and the corresponding simple rule:

$$\frac{C, T <: F_T; \Gamma \vdash e : S}{C; \Gamma \vdash \ \mathbf{type}\ T <: F_T\ \mathbf{in}\ e : S}\ \text{TypeDef}_{\text{Rec}}$$

and similarly for $=$ and $>:$.

2.9.3 Use cases of negation types

Negation types, which we denote `~A` for a type `A`, have been studied in type system designs [18, 58, 73], but have not been as widely adopted as union and intersection types. One may think that negation types typically bear little computational information that could ensure safety: What can be done with a value that is known to be "anything but a string" that could not be done on a string? Hence, the utility of negation types resides more in the value of types as a contract between an implementation and the user. Nonetheless, there are some specific types whose negation does bear computational value, such as

- `~Null`, the type of non-null values
- `~(Any ⇒ Nothing)`, the type of values that are not functions
- `~{bar:Any}`, the type of records that do not define the `bar` field, allowing it to be safely introduced

Flow types in pattern matching Consider the following program snippet:

```
type A
type B extends A
type  ... extends A

def handleB = λ x:A.
  match x with
    x:B ⇒ "good",
    _ ⇒ x
```

Without negation types, a language such as Scala may infer `String|A` as the result type of `handleB1`, even though its semantics of pattern matching allows overlapping patterns and prescribes a top-down execution of cases. A system based on flow typing could thus apply the fact that when `y` is returned, it does not have type `B1`, which suggests that a more precise return type for `handleB1` is `String | (A & ~B1)`.

MLscript [58] and its core subset MLstruct [73] offer negation types as well as union and intersections to assign more general types to expressions in a way similar to the above example, as witnessed by the following illustration:

```
def flatMap2 f opt =
  case opt of
    Some → f opt.value ,
    _    → opt
flatMap2 : ∀ A, B. (A → B) → (Some[A] | (~Some[Any] & B)) → B
```

In this example, the negation type allows excluding from `B` types for which `opt` would be caught by the first case of the match pattern, but with a different type parameter.

CDuce [18, 19] also uses negation types to assign more precise types to expressions. A simple example is the `not` function:

```
let not = fun x → if x then false else true
```

If `Falsy` is the type of values considered as wrong[3] it can be assigned the type

```
~Falsy → false & Falsy → true
```

which is more precise than `bool → bool`.

Unambiguous overload Function overloading allows defining different implementations of the same function based on the type of the value it is applied to. However, determining which implementation should be invoked can be ambiguous. Suppose `AB <: A` and `AB <: B`, and consider two function definitions

```
def f(x: A)
def f(x: B)
```

Now if `ab: AB`, resolving which `f` should `f(ab)` use is ambiguous. Such cases are typically rejected by compilers, or are otherwise resolved by a fixed notion of priority relying on precision or scope. Negation types allow giving arbitrary priority for such cases, by defining instead (if the first implementation should have priority)

```
def f(x: A)
def f(x: B & ~A)
```

Such use of negation thus enables expressive compile-time disambiguation of calls depending on the arguments they handle.

In general, negation provides a rich language to express disjointness of combinations of types. We thus also expect it to be of interest for match types in Scala [10], which impose disjointness conditions on cases when matching on the type.

[3] For example in JavaScript/TypeScript `false`, `""`, `0`, `-0`, `0n`, `undefined`, `null`, and `NaN`

2.9.4 Record types

Records are ubiquitous in programming languages, providing containers for data, and with subtyping of record types providing a theoretical foundation for class inheritance in object-oriented programming. Luckily, record types are directly representable in a free type lattice without any additional construct.

Definition 2.9.8 (Record types)**.** Fix some arbitrary set of labels a, b, ... and let

$$R_a[+A], R_b[+A], ... \in \mathcal{B}$$

be distinguished constant type constructors, each with one covariant argument. Additionally, let

$$\texttt{select}_a, \texttt{select}_b, ... \in \mathbb{L}$$

be constant symbols, each of type $\forall A.R_i[A] \to A$. Finally, for any finite set of labels $a_1, a_2, ...$ let

$$\texttt{build}_{a_1,...,a_n} \in \mathbb{L}$$

be a constant symbol of type $\forall A_1, \ldots, A_n.A_1 \to ... \to A_n \to R_{a_1}[A_1] \wedge ... \wedge R_{a_n}[A_n]$.

To adopt more familiar notation, we write

$$\{a_1 : A_1, \ldots, a_n : A_n\}$$

in place of $R_{a_1}[A_1] \wedge ... \wedge R_{a_n}[A_n]$.

Example 2.9.9. The following is an expression:

```
def x : {a : Int, b : String} := build_ab(2, "hello") in
      select_a[Int](x)
```

It is easy to show that our embedding of record types satisfies the expected subtyping relation of record types.

Lemma 2.9.10 (Width subtyping)**.**
For any labels $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$, for any types $A_1, \ldots, A_m, B_1, \ldots, B_n$

$$\frac{}{\{a_1 : A_1, \ldots, a_m : A_m, b_1 : B_1, \ldots, b_n : B_n\} <: \{a_1 : A_1, \ldots, a_m : A_m\}}$$

is a deducible OL^+ rule.

Proof. By definition the statement is equivalent to

$$R_{a_1}[A_1] \wedge ... \wedge R_{a_m}[A_m] \wedge R_{b_1}[B_1] \wedge ... \wedge R_{b_n}[B_n] \leq R_{a_1}[A_1] \wedge ... \wedge R_{a_m}[A_m]$$

which is a lattice tautology and hence provable. □

Lemma 2.9.11 (Permutation subtyping)**.** For any permutation of the same set of labels $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ and types $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$

$$\frac{}{\{a_1 : A_1, \ldots, a_n : A_n\} <: \{b_1 : B_1, \ldots, b_n : B_n\}}$$

is a deducible OL^+ rule.

Proof. This is again a lattice tautology by the commutativity and associativity rules. □

Lemma 2.9.12 (Depth subtyping)**.** For any set of labels $a_1, \ldots, a_n$ and types $A_1, \ldots, A_n$, $B_1, \ldots, B_n$

$$\frac{C \vdash A_1 <: B_1 \qquad \ldots \qquad C \vdash A_n <: B_n}{\{a_1 : A_1, \ldots, a_n : A_n\} <: \{a_1 : B_1, \ldots, a_n : B_n\}}$$

is a deducible OL^+ rule.

Proof. Follows from the application of the f-rule for each R_i. □

2.9.5 Prototype implementation and examples

As a proof of concept, we implemented a prototype type checker and interpreter for the language defined in Definition 2.9.2, with some syntactic sugar. The example in Listing 2.14 extends the simple example from earlier. The example in Listing 2.15 shows how union types, constrained polymorphism and equirecursive types interact to define a `reduce` function that can reduce arbitrarily nested lists of integers.

Listing 2.14: Example of a program using the OL type system to define `ntimes`

```
def ntimes[A where A <: Semigroup[A]] : A ⇒ Int ⇒ A ⇒ A :=
  \base: A. \n: Int. \elem: A.
    if[A] (eq[Int] n 0) then
      base
    else
      (combine[A] elem (ntimes[A] base (sub n ) elem))
in

def mult: Int ⇒ Int ⇒ Int := \x: Int. \y: Int. ntimes[Int] 0 x y
in

def repeat: Int ⇒ String ⇒ String := \n: Int. \s: String.
  ntimes[String] "" n s
in

def factorial: Int ⇒ Int :=
  \n: Int. if[Int] (lt n 2) then  else (mult n (factorial (sub n )))
in
print (repeat 3 "ha") // = "hahaha"
print (factorial 5) // = 20
```

Listing 2.15: Example of a program using the OL type system to define `reduce`

```
type LL where LL = Int | List[LL] in
def reduce[A, B where A <: Semigroup[A], B <: A | List[B]] : B ⇒ A :=
  \x: B. match x with
    y: A ⇒ y,
    z: List[B] ⇒ if (isEmpty[B] (tail[B] z))
      then (reduce[A, B] (head[B] z))
      else (combine[A]
        (reduce[A, B] (head[B] z))
        (reduce[A, A | List[B]] (tail[B] z))
      )
in
def nested: LL = List[LL](List[LL](, 2, 3), 0, List[LL](4, 5)) in
print (reduce[Int, LL] nested) // = 25
```

2.9.6 Union and intersection types in mainstream programming languages

Subtyping with union and intersection types has been implemented in several programming languages, but typically exhibits poor performance, incompleteness or even unsoundness. Moreover, their theoretical foundations are often not clearly stated, based on a possibly complex implementation itself rather than on simple axiomatic descriptions that implemented algorithms should follow. To more concretely illustrate the potential real-world impact that orthologic-based subtyping can have, we highlight three programming languages that were deployed on large codebases and that support subtyping with union and intersection types: Scala, TypeScript and Flow.

Scala Scala 3 supports union and intersection types with the laws of a lattice. Laws V1-V4 can be proven using the type checker, as can their duals, V1'-V4': given unconstrained types `A`, `B`, `C`, the Scala version 3.6.3 type checker accepts all the following:

```
1 summon[(A|B) =:= (B|A)]              summon[(A & B) =:= (B & A)]
2 summon[(A|A) =:= A]                  summon[(A & A) =:= A]
3 summon[(A|(B|C)) =:= ((A|B)|C)]      summon[(A & (B & C)) =:= ((A & B) & C)]
4 summon[(A|(A&B)) =:= A]              summon[(A & (A|B)) =:= A]
```

Here `LHS =:= RHS` is a Scala expression that invokes two subtyping checks between LHS and RHS, succeeding (with an evidence object) if they both succeed. Furthermore, using Scala's `Nothing` for the least element $\bot$ and `Any` for $\top$, the equations V5-V6 and V5'-V6' also hold:

```
1 summon[(A | Any) =:= Any]            summon[(A & Nothing) =:= Nothing]
2 summon[(A | Nothing) =:= A]          summon[(A & Any) =:= A]
```

Note that, in some cases, the Scala type system does not reason completely with assumptions, missing an opportunity to apply the transitivity of subtyping as well:

```
1 type A
```

```
type C
def foo[B >: A <: C](x: A) : C = x     // does not compile
def bar[B >: A <: C](x: A) : C = (x:B) // compiles, thanks to the x:B hint
```

Our algorithm supports reasoning with assumptions (axioms). Maintaining a global store of such assumptions and using our algorithm would eliminate such a source of incompleteness.

Going beyond the axioms of lattices, Scala also satisfies distributivity laws, accepting

```
summon[((A | B) & C) =:= ((A & C) | (B & C))]
summon[((A & B) | C) =:= ((A | C) & (B | C))]
```

We next consider performance consequences of distributivity in a type checker such as one in Scala (or, as we will see, TypeScript). Consider the families of types S_n and T_n indexed on even numbers defined as

```
S₂ = X₁ | X₂                          T₂ = X₂ | X₁
Sₙ₊₂ = Sₙ & (X₂ₙ₋₁ | X₂ₙ)             Tₙ₊₂ = Tₙ & (X₂ₙ | X₂ₙ₋₁)
```

That is, S_n and T_{n-1} only differ in the permutation of the disjuncts. Interestingly, the Scala compiler fails to prove `Sₙ =:= Tₙ` for as small as $n = 6$. Moreover, the compiler runtime increases exponentially despite the incompleteness, with type checking time for $n = 26, 28, 30, 32, 34$ being $13, 41, 135, 280, 640$ seconds. In contrast, our verified implementation of orthologic proof search solves such formulas in milliseconds (Subsection 2.12.2), exhibiting quadratic time.

As of Scala 3.7, the type checker also accepts the following "constructor distributivity" law : if `T[+X]` is a covariant type constructor, then `T[A] & T[B] =:= T[A & B]`. Alarmingly, we recently discovered that this rule is unsound, leading to a runtime error for programs that type check, such as this one:

```
trait L[+A]{val a:A}
trait R[+B]{val b: B}
class LR(val a: Int, val b: String) extends L[Int] with R[String]
type E[+A] = L[A] | R[A]

val x: E[Int] & E[String] = LR(4, "hi")
val y: E[Int&String] = x
val z: Int&String = y match
  case l : L[Int&String] ⇒ l.a
  case r : R[Int&String] ⇒ r.b
z.toUpperCase
```

Following our report, this has been removed from future Scala versions. Further study is necessary to establish when such laws are sound, and our approach does not consider such problematic laws. Our constructors are simply monotonic (covariant), antimonotonic (contravariant), or uninterpreted (invariant) in their arguments. The algorithms we present support efficient and complete reasoning about subtyping (and equivalence) and normalization for this model.

TypeScript TypeScript adds static types and type checking to JavaScript. It supports union and intersection types, which form a distributive lattice. Analogously to Scala, TypeScript validates the following assertions:

```
type Eq<T, U> = T|U extends T&U? true : false;
type Assert<T extends true> = T;

type V = Assert<Eq<A|B, B|A>>;
type V2 = Assert<Eq<(A|B)|C, A|(B|C)>>;
type V3 = Assert<Eq<A|A, A>>;
type V4 = Assert<Eq<A|(A&B), A>>;
type V5 = Assert<Eq<A|unknown, unknown>>;
type V6 = Assert<Eq<A|never, A>>;

type V_ = Assert<Eq<A&B, B&A>>;
type V2_ = Assert<Eq<(A&B)&C, A&(B&C)>>;
type V3_ = Assert<Eq<A&A, A>>;
type V4_ = Assert<Eq<A&(A|B), A>>;
type V5_ = Assert<Eq<A&never, never>>;
type V6_ = Assert<Eq<A&unknown, A>>;

type Distributivity = Assert<Eq<(A&B)|C, (A|C)&(B|C)>>
type Distributivity_ = Assert<Eq<(A&B)|C, (A|C)&(B|C)>>
```

On the families S_n and T_n we introduced in the previous section, TypeScript demonstrates significantly faster performance than Scala type checker, but still exhibits exponential behaviour, with type checking times for $n = 26, 28, 30, 32$ being $1.5, 2.1, 3.1, 5.5$ seconds, and interrupting compilation with an error TS2590 for $n \geq 34$, when the size of types after normalization exceeds a certain limit.

On the other hand, TypeScript does not support polymorphic functions with lower bounds as in the above Scala example, nor nominal subtyping declarations.

Flow Flow [28] is a type system for JavaScript based on flow typing. It also uses types that form a lattice, but it is *not* distributive. Moreover, the subtype checking algorithm is highly incomplete: it does not even accept $\Phi_4 = \Phi'_4$ without user-written assertions, that is, it fails to prove

```
X|X2 & X3|X4 = X2|X & X4|X3
```

Note the contrast with Scala and TypeScript, which implement distributivity and may compute a disjunctive normal form, which gives them a reason to be slow. Here, the type lattice of Flow is not claimed to be distributive, so there is no reason to expect slow performance: an approach based on lattices or ortholattices would yield a complete and fast subtyping algorithm.

Other languages have a lattice-like subtyping relation. For example, Julia, Typed Racket and Crystal offer union types to the user, but not intersection types. Kotlin uses intersection types internally, but does not expose them to the user. PHP offers union

and intersection types, but they cannot be combined. In general, we believe that algorithms based on lattices with constructors would benefit type checker implementations in these systems. In summary, Scala and TypeScript implement distributivity law, but face slowdown in type checking, whereas Flow does not implement distributivity. We therefore feel justified in not assuming the general distributivity law.

To conclude, orthologic subtyping provides a principled and theoretically-grounded basis for subtyping with union, intersection and negation types together with covariant and contravariant type constructors. This approach delegates all the semantic complexity of subtyping to a standalone decision procedure that is proven correct with respect to the mathematical specification. We showed that orthologic laws with axioms enable a variety of programming patterns that programmers have come to expect from high-level programming languages. By relying on the precisely defined theoretical grounds of orthologic proof theory, we believe we can avoid the pitfalls and maintenance issues that arise from ad-hoc additions of features. Hence, we hope that orthologic-based subtyping can enable future programming languages with simple core laws, efficient algorithms and expressive subtyping features that are clearly justified by the theory.

2.10 Verified orthologic in Rocq

Beyond the fascinating theoretical properties of orthologic, the main objective is to use it as an efficient and reliable building block in automated reasoning software such as SMT solvers, proof assistants and automated theorem provers. As the tools grow and building blocks compound, so does the possibility of an implementation error. To ensure that they can be used as trusted components in program verification pipelines, these verification algorithms should themselves be verified.

In this section, we discuss the formalization and verification of orthologic, using the Rocq proof assistant. We will start by formalizing the orthologic sequent calculus, including soundness and completeness with respect to ortholattices and the cut elimination theorem. Then, we will implement multiple versions of the proof search procedure for the case where there are no axioms, starting from a naive implementation of the decision procedure which has exponential complexity. To obtain a polynomial version, we need to leverage memoization, i.e. storing in a table the intermediate results of the recursive calls. Moreover, using structural equality to check if a key is in the memoization map costs an additional linear runtime factor. To obtain an optimal quadratic version, we modify the algorithm to use a form of reference equality. As our algorithm is purely functional, this means we have to extend our formulas' abstract syntax trees to assign to each node a unique identifier (or pointer) that can then be used in the memoization map. We will formally prove the correctness of these constructions.

We implement and verify multiple versions of the algorithm: without optimization ($\widetilde{\mathcal{O}}(2^n)$), with memoization using lists ($\widetilde{\mathcal{O}}(n^5)$), using AVL maps ($\widetilde{\mathcal{O}}(n^3)$), and using AVL maps and simulated reference equality ($\widetilde{\mathcal{O}}(n^2)$). Memoization and reference equality are generic and important in many tools and algorithms. The core of our techniques can generalize to any recursive algorithm over algebraic datatypes (ADT).

Finally, using the technique of proofs by reflection available in Rocq, we will obtain a set of executable proof tactics, each deciding equality modulo orthologic rules and applicable to any ortholattice. This includes in particular the type `bool` of boolean values. This tactic is able to solve automatically, for example,

```
true = (a && b) || (negb a) || (negb b)
```

As of 2026, all the tactics are available as a Rocq plugin along with benchmarks at

github.com/SimonGuilloud/orthologic-coq,

and on opam.

Acknowledgement of contributions All the contributions of this section are the result of joint work with Clément Pit-Claudel.

2.10.1 Formalizing ortholattices and orthologic

We first formalize the algebraic class of ortholattices with function symbols as a type class according to Definition 2.1.15. The corresponding Rocq definition is in Listing 2.16.

```
Class Ortholattice := {
  A : Set;
  leq : relation A where "x ⩽OL y" := (leq x y);
  meet : A → A → A where "x ∩ y" := (meet x y);
  join : A → A → A where "x ∪ y" := (join x y);
  neg : A → A where "¬ x" := (neg x);
  zero : A;
  one : A;
  app: Id → A → A → A → A → A;
  equiv: relation A where "x == y" := (equiv x y);
  equiv_leq : ∀ x y, (x == y) ⟷ ( (x ⩽OL y) ∧ (y ⩽OL x));
  zero_leq : ∀ x, (leq zero x);
  one_leq : ∀ x, (leq x one);
  ...
}.
```

Listing 2.16: Definition of an Ortholattice in Rocq.

(Anti)monotonic function symbols in OL^+ can in theory have arbitrary arity. But constructors of arbitrary arity are difficult to represent in Rocq so we make a simplification: we assume that all function symbols take two monotonic and two antimonotonic arguments. This is sufficient to simulate arbitrary arity: For example, a function symbol with one covariant argument and one invariant argument $f(x, y)$ can be represented as $f'(x, y, y, 0)$ while functions with more arguments can be represented using nested applications of binary functions. The `app` constructor is used to represent application of an arbitrary function symbol to some arguments.

We then implement using Ltac an extension of *Whitman's algorithm* [32], a simple decision procedure for lattices. This helps us to quickly show a number of useful lemmas about ortholattices but is not complete.

```
Ltac whitmanPlus := match goal with
  | [ ⊢ _ == _] ⇒ rewrite equiv_leq; split; whitmanPlus
  | [ ⊢ zero ⩽OL _] ⇒ apply zero_leq
  | [ ⊢ _ ⩽OL one] ⇒ apply one_leq
  | [ ⊢ ?x ⩽OL ?x] ⇒ apply P1
  | [ ⊢ _ ⩽OL _ ∩ _] ⇒ apply P6; whitmanPlus
  | [ ⊢ _ ∪ _ ⩽OL _] ⇒ apply P6'; whitmanPlus
  | [ ⊢ _ ∩ _ ⩽OL _] ⇒
    try (apply glb1; whitmanPlus; eauto; fail);
    try (apply glb2; whitmanPlus; eauto; fail)
  | [ ⊢ _ ⩽OL _ ∪ _] ⇒
    try (apply lub1; whitmanPlus; eauto; fail);
    try (apply lub2; whitmanPlus; eauto; fail)
  | [ ⊢ ¬ _ ⩽OL ¬ _] ⇒ apply P8; whitmanPlus
  | [ ⊢ (app ?f _ _ _ _) ⩽OL app ?f _ _ _ _] ⇒ apply frule; whitmanPlus
  | [ ⊢ _ ⩽OL _] ⇒ try (eauto;fail)
```

```
end.
Lemma example {OL: OLPlus} a b : (a ∩ b) ⩽OL (a ∪ b).
Proof. whitmanPlus. Qed.
```

This simple tactic is subsumed by the tactic for ortholattices we will obtain with reflection. The different cases of the tactic are restrictions of the deduction rules of $\mathbf{LO}^+$ (Definition 2.2.3). We then show that $=$, under the axiom `x = y ⟷ x ⩽ y ∧ y ⩽ x`, is a congruence relation for $\leqslant$, $\cap$, $\cup$, and $\neg$. This makes ortholattices *setoids*, and enables the use of *generalized rewriting* [85].

We define in the standard way the type of ortholattice terms, corresponding to $\mathcal{T}_{OL^+}(X)$ where $X = \{\texttt{Var}\ i \mid i \in \mathbb{N}\}$, and the evaluation of a term in an arbitrary ortholattice:

```
Inductive Term : Set :=
  | Var : positive → Term
  | Meet : Term → Term → Term
  | Join : Term → Term → Term
  | Not : Term → Term
  | App : Id → Term → Term → Term → Term → Term.

Fixpoint eval {OL: OLPlus}
  (f: positive → A)
  (t: Term)  : A :=
  match t with
  | Var n ⇒ f n
  | Meet t1 t2 ⇒ (eval f t1) ∩ (eval f t2)
  | Join t1 t2 ⇒ (eval f t1) ∪ (eval f t2)
  | Not t1 ⇒ ¬ (eval f t1)
  | App id ty1 ty2 tz1 tz2 ⇒
      app id (eval f ty1) (eval f ty2) (eval f tz1) (eval f tz2)
  end.
```

Here `positive` is the type of natural numbers represented in binary.

We continue with the description of $\mathbf{LO}^+$ (Definition 2.2.3), representing sequents as (ordered) pairs of annotated terms:

```
Inductive AnTerm : Set :=
  | L : Term → AnTerm
  | R : Term → AnTerm.
Definition Sequent (l r : AnTerm) := (l, r).
```

where `L` is a formula on the left and `R` is a formula on the right.

By implementing the proof system using dependent inductive types, the correctness of a proof is guaranteed by construction, and no additional proof-checking function is required. The proof system is itself parametrized by a list of axioms, each being a sequent. This allows to represent both the pure orthologic proof system (no axioms) and the more general case of orthologic with axioms.

In Definition 2.2.3, sequents are formally considered as sets. We have defined them in Rocq using ordered pairs and hence need to define an additional rule, simulating the

```
Inductive LOP {ax : Axioms}: AnTerm*AnTerm → Set :=
  | Hyp : ∀ {p}, LOP (L (Var p), R (Var p))
  | Contract : ∀ {gamma} {delta}, LOP (gamma, gamma) →
      LOP (gamma, delta)
  | LeftAnd1 : ∀ {a} {b} {delta}, LOP (L a, delta) →
      LOP (L (Meet a b), delta)
  | LeftAnd2 : ∀ {a} {b} {delta}, LOP (L b, delta) →
      LOP (L (Meet a b), delta)
  | LeftOr : ∀ {a} {b} {delta}, LOP (L a, delta) → LOP (L b, delta) →
      LOP (L (Join a b), delta)
  | LeftNot : ∀ {a} {delta}, LOP (R a, delta) →
      LOP (L (Not a), delta)
  | RightAnd : ∀ {a} {b} {gamma}, LOP (gamma, R a) → LOP (gamma, R b) →
      LOP (gamma, R (Meet a b))
  | RightOr1 : ∀ {a} {b} {gamma}, LOP (gamma, R a) →
      LOP (gamma, R (Join a b))
  | RightOr2 : ∀ {a} {b} {gamma}, LOP (gamma, R b) →
      LOP (gamma, R (Join a b))
  | RightNot : ∀ {a} {gamma}, LOP (gamma, L a) →
      LOP (gamma, R (Not a))
  | FRule : ∀ {id} {y1 y2 z1 z2} {y1' y2' z1' z2'},
      LOP (L y1, R y1') → LOP (L y2, R y2') →
      LOP (L z1', R z1) → LOP (L z2', R z2) →
      LOP (L (App id y1 y2 z1 z2), R (App id y1' y2' z1' z2'))
  | \Swap : ∀ {gamma} {delta}, LOP (gamma, delta) →
      LOP (delta, gamma)
  | Axio : ∀ {s}, In s ax → LOP s
  | Cut : ∀ {gamma} {b} {delta}, LOP (gamma, R b) → LOP (L b, delta) →
      LOP (gamma, delta).
```

Listing 2.17: Definition of the orthologic proof system $\mathbf{LO}^+$ in Rocq.

set-like nature of sequents: the Swap rule.

$$\frac{\Gamma, \Delta}{\Delta, \Gamma}\ \textsc{Swap}$$

The rules of the proof system are directly translated from Definition 2.2.3 to Rocq as constructors of the inductive type `LOP` in Listing 2.17.

Soundness (Theorem 2.2.5) and completeness (Theorem 2.2.6) follow their respective paper proofs.

```
Theorem soundness (ax : Axioms) l r:
  (@LOP ax (l, r)) → anTerm_leq ax l r.

Theorem completeness (ax : Axioms) l r :
  term_leq ax l r → has_proof ax l r.
```

In particular to prove completeness, we show that the type `Term` is itself an ortholattice:

```
#[refine] Instance TermOL ax : OLPlus := {
  A := Term;
  leq := has_proof ax;
  meet := Meet;
  join := Join;
  neg := Not;
  zero := Zero;
  one := One;
  app := App;
  equiv := fun x y ⇒ (has_proof ax x y) ∧ (has_proof ax y x);
}.
```

2.10.2 Formalization of cut elimination

We are now ready to prove the cut elimination theorem for orthologic (Theorem 2.2.10), which states that any sequent provable in $\mathbf{LO}^+$ has a cut-free proof, that is a proof in $\mathbf{CF}^+$ (Definition 2.2.9).

```
Theorem LOP_to_CFP gamma delta : LOP (gamma, delta) → CFP (gamma, delta).
```

This theorem is not straightforward to formalize. The key part, as in the paper proof, is to show that the Cut rule is admissible. Once this is done, we can complete the proof by double induction, first on the number of instances of the Cut rule appearing in a proof. The corresponding lemma takes as arguments proofs that the given fuel is larger than the metric it represents, and every induction step needs to justify that the measures are decreasing. This step allows to reduce the problem to admissibility of the Cut rule in $\mathbf{CF}^+$.

```
Lemma inner_cut_elim : ∀
  (fuelB: positive)
  (b: Term) (good_fuelB: fuelB ⩾ term_size b)
  (fuelSize: positive)
  (gamma: AnTerm) (delta: AnTerm)
  (A: CFP (gamma, R b))
  (B: CFP (L b, delta))
  (good_fuelSize: fuelSize ⩾ (proof_size' A + proof_size' B)),
  CFP (gamma, delta).
```

The main challenge of the proof comes from the sheer size of the case analysis: the Swap step essentially duplicates every other proof step by allowing them to act on the first or second formula, which implies we have to analyse each of $\mathcal{A}$ and $\mathcal{B}$ on 23 cases each, for a total of more than 500 cases. In practice, we can first do the analysis on $\mathcal{A}$, and for some cases the proof is independent of the structure of $\mathcal{B}$, as in case 3 of the proof of Theorem 2.2.10. However, there is no way to undo the case analysis on $\mathcal{A}$ when conversely the cases of pattern matching on $\mathcal{B}$ have a proof independent of the structure of $\mathcal{A}$.

Then, some combinations of cases are impossible. For example, it is not possible for $\mathcal{A}$ to conclude with a `RightAnd` and $\mathcal{B}$ with `LeftOr1`, as the cut formula b would then

need to be both a conjunction and a disjunction. Thanks to the use of dependent types to define the proof system, those cases are automatically eliminated. In the proof of Theorem 2.2.10, we used a lot of reasoning by symmetry, which cannot be easily done in Rocq. In the end, the formal proof contains around 200 cases.

We also define a cut-free proof system for bounded lattices, `CFPBLPlus`. Cut elimination is proven the same way as for ortholattices.

2.10.3 Formalization of normalization

We then formalize the BL^+ and OL^+ normalization algorithms from Section 2.4.

Infrastructure First, we formalize a representation of terms in bounded lattices where meets and joins are multiary:

```
Inductive FlatTerm: Set :=
| FVar (v: nat)
| FMeet (args: list FlatTerm)
| FJoin (args: list FlatTerm)
| FApp (f: Id) (y1 y2 z1 z2: FlatTerm)
| FZero
| FOne.
```

This is useful to express the η and ζ functions from Subsection 2.4.1, but requires significant boilerplate and explicit handling of lists. In particular, we define the core notion of equivalence of flat terms `eq_flat` as equality modulo reordering.

We then implement a proof system `FlatCFBLPlus` that is equivalent to $\mathbf{CF}^+_{\boldsymbol{BL}}$ but where the left and right rules for meets and joins are replaced by a single rule for multiary meets and joins. We show that it is sound and complete by reducing it to `CFPBLPlus` and prove useful inversion lemmas.

η and ζ both require computing the $\leq_{BL^+}$ relation. We implement this relation as a decision procedure, following Whitman's algorithm, and prove its correctness using the completeness of `CFPBLPlus`.

We then define what it means for a term to be in minimal form:

```
Definition is_blp_min_size (ft: FlatTerm) : Prop :=
  forall fs, ft ∼BL+ fs → fsize ft ⩽ fsize fs.
```

and then show the characterization of normalization for bounded lattices as described in Theorem 2.4.4.

The ζ and η functions. The next step is to define `Zeta` and `Eta` as in Subsection 2.4.1, show that they preserve equivalence, and that they produce terms satisfying the characterization of minimal forms. We can then define the normal form function:

```
Definition normalize_term (t: Term) : FlatTerm :=
  Eta (Zeta (flatten t)).
```

We prove that it satisfies all necessary properties through a large list of intermediate lemmas. We finally obtain the core correctness theorem for BL^+ normalization:

```
Theorem blp_term_normalization : forall t,
  (* 1. Correctness: input is BLP-equivalent to decoded output *)
  term_bequiv t (flat_to_term (normalize_term t)) ∧
  (* 2. Canonicity: BLP-equivalent inputs produce eq_flat-equal outputs *)
  (forall t', term_bequiv t t' →
    eq_flat (normalize_term t) (normalize_term t')) ∧
  (* 3. Minimality: output has minimal size among BLP-equivalent terms *)
  blp_minimal_form (flat_to_term (normalize_term t)).
```

Orthologic normalization. Neither the pseudo-nnf nor the β function from Subsection 2.4.2 require flattened terms, so we work directly with binary orthologic terms. This greatly reduces boilerplate. We define pseudo-nnf and a dedicated representation for $\mathcal{T}_{\overline{BL^+}}(X \cup X')$:

```
Inductive DeltaTerm : Set :=
  | DVar (v: nat)
  | DNegVar (v: nat)
  | DMeet (a b: DeltaTerm)
  | DJoin (a b: DeltaTerm)
  | DApp (f: nat) (y1 y2 z1 z2: DeltaTerm)
  | DNegApp (f: nat) (y1 y2 z1 z2: DeltaTerm)
  | DZero
  | DOne.
```

and corresponding helper lemmas, including redefinition of $\leq_{BL^+}$ on `DeltaTerm` and its decidability. We also redefine a dedicated proof system for `DeltaTerm` corresponding to $\mathbf{CF}^+_\delta$, and again show appropriate soundness and completeness results.

Then, we define the β function, and show its key property corresponding to Lemma 2.4.11:

```
Lemma beta_equiv_bridge : forall ft1 ft2,
  (beta ft1) ≤OL+ (beta ft2) ⟷
  (beta ft1) ≤BL+(beta ft2).
```

Finally, we show correctness of the normalization procedure for orthologic:

```
Theorem olp_normalization_correct : forall t,
  (* 1. Correctness: input is OLP-equivalent to decoded output *)
  (t ≤OL+ (flat_to_olp (normalize_olp t)) ∧
    (flat_to_olp (normalize_olp t)) ≤OL+ t) ∧
  (* 2. Canonicity: OLP-equivalent inputs produce eq_flat-equal outputs *)
  (forall t', t ≤OL+ t' → t' ≤OL+ t →
    eq_flat (normalize_olp t) (normalize_olp t')) ∧
  (* 3. Minimality: output has minimal size among OLP-equivalent terms *)
  olp_minimal_form (flat_to_olp (normalize_olp t)).
```

```
Fixpoint decideOL_base (fuel: nat) (g d: AnTerm) : bool :=
  match fuel with
  | 0 ⇒ false
  | S n ⇒
    match (g, d) with
    | (L (Var a), R (Var b) )  ⇒ (Pos.eqb a b) | _ ⇒ false (* Hyp *)
    end || (
    decideOL_base n g g || ( (* Replace *)
    match g with
    | L (Meet a b) ⇒ decideOL_base n (L a) d | _ ⇒ false (* LeftAnd1 *)
    end || (
    match g with
    | L (Meet a b) ⇒ decideOL_base n (L b) d | _ ⇒ false (* LeftAnd2 *)
    end || (
    match g with
    | L (Join a b) ⇒ decideOL_base n (L a) d &&
                     decideOL_base n (L b) d | _ ⇒ false (* LeftJoin *)
    end || (
    match g with
    | L (Not a) ⇒ decideOL_base n (R a) d | _ ⇒ false (* LeftNot*)
    end || (
     ... (* Symmetric right cases *)
     || (
    decideOL_base n d g (* \Swap *)
    )))))))
  end.
```

Listing 2.18: Decision procedure for the *word problem for ortholattices*.

2.10.4 Decision procedure for orthologic in Rocq

Next, we formalize a decision procedure for the word problem for ortholattices using a variant of the proof search procedure from Theorem 2.2.12 in the special case where there are no axioms and no function symbols. In this case, we can proceed by backward proof search, starting from the sequent to prove and applying the rules of $\mathbf{LO}^+$ backward until we reach axioms. First, we implement a naive version that does not memoize the results of recursive calls. This algorithm has superexponential runtime complexity and is of no practical use, but it is simpler to implement and prove correct and it will allow us to prove the correctness of the more complex versions relative to it. This implementation is in Listing 2.18.

The soundness of this algorithm is expressed as

```
Theorem decideOL_base_correct  :
  ∀ n g d,
  (decideOL_base n g d) = true →
  ∃ _: (LOP (g, d)).
```

Reflection

Suppose we have a goal $s \leq t$ or $s = t$, where s and t are expressions in an arbitrary ortholattice, for example:

```
Lemma example a b: a && negb a = negb b && (a && b).
```

We want to solve this goal by executing the above algorithm, which we proved correct, using the technique of proof by reflection. First, we split $s = t$ into two independent inequalities. Then, reflection consists in two separate steps: *reification* and *evaluation* (or type checking).

Reification consists in finding two terms `s'` and `t'` as well as a function `f: positive → bool`[4] such that `eval s' f` is convertible to `s` and `eval t' f` is convertible to `t`. Here, convertible means that the two expressions are equivalent up to $\beta\alpha$ conversion, unfolding of definitions, and additional relations defined by Rocq[5]. In particular, any expression can always be freely replaced by one it is convertible to, without additional proof. Hence, by definition of `eval`, the inequality in the goal is convertible to (and can be changed to):

```
eval ((Var 0) ∩ ¬ (Var 0)) f       ⩽
eval ¬( (Var 1) ∩ ((Var 0) ∩ (Var 1))) f.
```

We can then apply a lemma derived from the soundness of the decision procedure with respect to the proof system, and of the proof system with respect to the class of ortholattices to reduce the problem to:

```
decideOL (L ((Var 0) ∩ ¬ (Var 0)))
         (R ¬( (Var 1) ∩ ((Var 0) ∩ (Var 1))))
    = true.
```

Now, we can complete this proof using the term `eq_refl bool true : (true = true)`, which is by definition a proof of `true = true`. For this to type check, the left-hand side of the goal equality needs to be convertible to `true`. Hence to type check the proof, Rocq *evaluates* the algorithm. Effectively, we have offloaded the burden of proof to the evaluator inside Rocq's kernel. Note also that the proof has constant size.

We now describe how to find a suitable `f` in our reification process, which has to be implemented in its own tactic. First, collect all the leaves of s and t in a list named `env`. A leaf is a subexpression that is neither a meet, a join nor a negation. These leaves will correspond to variables in `s'` and `t'`. f is then the function that maps n to the element in the list at place n. The reification tactic then uses this list to compute the terms `s'` and `t'`. We can mechanize the entire process as a tactic `solveOL`, solving, for example:

```
Lemma example a b:
  a && negb a = negb b && (a && b).
Proof. solveOL BoolOL. Qed.
```

[4] In general, `f: positive → OL`, where `OL` is the ortholattice in which the terms live.
[5] See *https://rocq-prover.org/doc/v8.18/refman/language/core/conversion.html*.

Note on soundness and completeness An error of implementation in a decision procedure can take two forms: soundness (incorrectly accepting a wrong equality) and completeness (incorrectly rejecting a correct equality). However, Rocq's logical kernel guarantees the soundness of every tactic. In the process above, the decision algorithm is proven sound. However, the reification process is not and cannot be, so it may contain errors and produce incorrect terms. In that case however, Rocq's kernel will refuse to replace the goal and produce an error message.

Verified memoization

The naively implemented proof search procedure has super-exponential runtime complexity, making it unusable in practice. We need to optimize it. First, observe that some proof steps, when applied backward, are not merely sufficient conditions but also necessary. For example, consider a sequent of the form $\Gamma, (\phi_1 \wedge \phi_2)^L$. It is a theorem of lattices that $s \leq t_1 \wedge t_2 \iff s \leq t_1 \;\&\; s \leq t_2$, corresponding to the RightAnd step. Hence, for $\Gamma, (s \wedge t)^R$ to have a proof, it is not only sufficient but also necessary that both Γ, s^R and Γ, t^R have a proof. It follows that if RightAnd is applicable, we do not need to try other steps, and similarly with LeftOr, Hyp, RightNot and LeftNot. Moreover, note that the algorithm will naively try to apply the Swap rule over and over. This makes the algorithm structurally non-terminating, and only fuel prevents infinite loops. This can be prevented by "unfolding" the swap step, and ensuring that a meaningful rule is always applied, but loops also are possible using the Replace rules. To handle this situation, we add two boolean variables to the input of the algorithm, each one denoting respectively whether `(g, g)` and `(d, d)` have already been tried. We implement a second version of the algorithm, named `decideOL_opti` implementing these optimizations:

```
Fixpoint decideOL_opti(fuel: nat)(g d: AnTerm)(cg cd: bool): bool:=
  match fuel with
  | 0 ⇒ false
  | S n ⇒ match (g, d) with
    ...
  end.
```

We prove its soundness and implement a corresponding reflection tactic `solve_OL_opti`.

The algorithm still has an exponential runtime of $\mathcal{O}(2^n)$, but always terminates without relying on the fuel. To obtain a polynomial time procedure, we need to implement *memoization*. Remember that thanks to the subformula property (Theorem 2.2.11), the algorithm only ever sees at most $\mathcal{O}(n^2)$ different sequents built from subformulas of the input. Using memoization, we can ensure that the body of the program is never executed more than $\mathcal{O}(n^2)$ times. Note that we also need to memoize the two boolean flags `cg` and `cd`, but this only multiplies the number of inputs by 4.

This version of the algorithm can be implemented using the *state monad* paradigm. Let `MemoMap` be some type of maps with keys in `(AnTerm * AnTerm)`

and values in `bool`. The function `decideOL_memo` then returns an object of type `MemoMap → (bool, MemoMap)`. Boolean conjunctions and disjunctions are modified to compute the results sequentially, so that when computing `a | b|`, where `a` and `b` are each a different computation, `b` is only computed if `a` returned `false`, and the memoization map is passed along.

```
Definition mor (left : MemoMap → (bool * MemoMap))
               (right : MemoMap → (bool * MemoMap))  :=
  fun m ⇒ match left m with
  | (true, m) ⇒ (true, m)
  | (false, m) ⇒ right m
end.

Infix "|||" := mor (at level 60).
```

and the decision algorithm is modified as follows:

```
Fixpoint decideOL_memo (fuel: nat)
      (g d: AnTerm) (cg cd: bool) (memo: MemoMap) : (bool * MemoMap) :=
  match find (g, d, cg, cd) memo with
  | Some (_, b) ⇒ (b, memo)
  | None ⇒ (match fuel with
    | 0 ⇒ (false, memo)
    | S n ⇒ let (b, m) :=
    (match (g, d) with
      | (L (Join a b), _) ⇒ decideOL_memo n (L a) d false cd &&&
                            decideOL_memo n (L b) d false cd
      | (_, L (Join a b)) ⇒ decideOL_memo n g (L a) cg false &&&
                            decideOL_memo n g (L b) cg false
      | (R (Meet a b), _) ⇒ decideOL_memo n (R a) d false cd &&&
                            decideOL_memo n (R b) d false cd
      | (_, R (Meet a b)) ⇒ decideOL_memo n g (R a) cg false &&&
                            decideOL_memo n g (R b) cg false
      | (L (Not a), _) ⇒ decideOL_memo n (R a) d false cd
      | (_, L (Not a)) ⇒ decideOL_memo n g (R a) cg false
      | (R (Not a), _) ⇒ decideOL_memo n (L a) d false cd
      | (_, R (Not a)) ⇒ decideOL_memo n g (L a) cg false
      ... (* other cases*)
      end) memo
      in
        (b, ((g, d, cg, cd), b) :: m) end)
  end.
```

The correctness of the algorithm is expressed as equivalence with the non-memoized version of the algorithm, and relative to the correctness of the map given as input. A map is correct if and only if for every key k with a value of `true`, the non-memoized version of the algorithm returns `true` on k.

```
Definition memomap_correct (l: MemoMap) :=  ∀ g d cg cd,
  match find (g, d, cg, cd) l with
  | Some (_, true) ⇒ ∃ n,  (decideOL_opti n g d cg cd = true)
  | _ ⇒ True
  end.
```

In particular, the empty map is correct. Note that the two versions of the algorithm are not necessarily equivalent for an arbitrary allowance of fuel: the memoized version might require less. We prove the soundness of the memoized algorithm[6]:

```
Theorem decideOL_memo_correct :
  ∀ n g d cg cd l,
  (memomap_correct l) →
  (memomap_correct (snd (decideOL_memo n g d cg cd l))) ∧
  ( ((fst (decideOL_memo n g d cg cd l)) = true) →
    ∃ n0, (decideOL_opti n0 g d cg cd) = true ).
Proof.
```

The induction has to be performed jointly on the statement that the returned map is correct, and that the returned truth value is the same as that of the original algorithm. The proof again proceeds with a large case analysis, which is best performed using specialized automation, and side lemmas about the monadic structure of the memoization. On the other hand, the proof is completely independent of the orthologic proof system and would work similarly with any other recursive algorithm on algebraic datatypes.

In one version of the algorithm, the memoization map is implemented as a list of pairs. However, lookup inside a list takes time linear in its size. As explained above, the number of stored values is quadratic in the size of the input. Moreover, checking equality between the input and the elements of the list costs an additional linear factor. Overall, the complexity of lookup is cubic, for a total runtime of $\mathcal{O}(n^5)$.

Instead, we can use AVL maps from the Rocq standard library. This requires to define a total order on `anTerm`, which we do following the usual total order on labelled ordered trees. Sorted maps such as AVL maps only require a logarithmic number of comparisons, down from linear with a list-based implementation. However, comparison still takes linear time, for a total time complexity of $\mathcal{O}(n^3 \log n)$.

We hence obtain two more versions of the algorithm, `decideOL_memo` and `decideOL_fmap`, and two corresponding tactics `solveOL_memo` and `solveOL_fmap`.

2.10.5 Reference equality

Lookup in a map requires deciding either equality or ordering between two terms, which takes time linear in the size of the terms and even in a hash map, equality checking is necessary to avoid collisions. In the previous two algorithms, either based on lists or on AVL maps, equality is structural: two terms are equal if they have the same constructor and recursively their arguments are equal. However, this is *too strong* for memoization.

As in Section 2.3, observe that if we replace this notion of equality by a strictly

[6]Note that completeness is not required to define a reflection-based tactic.

weaker relation, the algorithm is still sound, and we only risk losing some of the benefits of memoization.

Then recall that by the subformula property, the algorithm only ever sees the $\mathcal{O}(n^2)$ different subnodes of the original input. Assigning a different binary identifier to each of these nodes requires only $\mathcal{O}(\log(n))$ bits for each identifier. Then, if two terms have the same identifier, they must be structurally equal.

This corresponds, in imperative programming, to *pointer equality*. Checking if two objects have the same location in memory is a sound approximation to deciding if they are structurally equal. Formally, we define an extended version of the datatype of terms:

```
Inductive TermPointer : Set :=
  | VarP : positive → Pointer → TermPointer
  | MeetP : TermPointer → TermPointer → Pointer → TermPointer
  | JoinP : TermPointer → TermPointer → Pointer → TermPointer
  | NotP : TermPointer → Pointer → TermPointer.
```

where `Pointer := positive` is the type of binary positive numbers. We define the projection onto regular terms `ForgetPointer : TermPointer → Term` and a getter `GetPointer : TermPointer → Pointer` as expected. We extend pointers to annotated pointers:

```
Inductive AnPointer : Set :=
  | NP : AnPointer
  | LP : Pointer → AnPointer
  | RP : Pointer → AnPointer.
```

and `TermPointer` to `AnTermPointer` similarly. For `g` an `AnTermPointer` (that is, a term with a pointer and a left or right annotation), we denote `[[g]]` its corresponding `AnPointer`. We again use AVL maps, but this time with keys being pairs of `AnPointer`. The algorithm is modified accordingly:

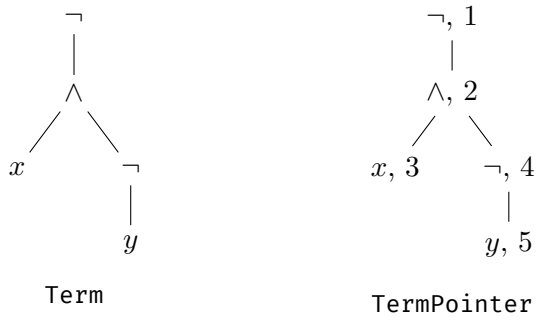
```
Fixpoint decideOL_pointers
    (fuel: nat) (g d: AnTermPointer)
    (cg cd: bool) (memo: MemoMap)
    : (bool * MemoMap) :=
  match M.find ([[g]], [[d]], cg, cd) memo with
  | Some b ⇒ (b, memo)
  | None ⇒ (match fuel with
    | 0 ⇒ (false, memo)
    | S n ⇒ let (b, m) := ... in
      (b, AnPointerPairAVLMap.add ([[g]], [[d]], cg, cd) b m)
    end)
  end.
```

As a reminder, `fuel` is an accessory argument ensuring termination, `g` and `d` are the actual input formulas augmented with pointers, `cg` and `cd` are flags used to prevent loops and `memo` is the memoization map. In practice, we define a separate function `decideOL_pointers_simp g d`, setting initial `fuel` to $(|g| + |d|)^2$, `cg` and `cd` to false, `memo` to the empty map and computing the pointers for `g` and `d`.

The correctness of the algorithm of course depends on how pointers are assigned. If two different terms are assigned the same pointer, the algorithm will not be correct. Formally, `GetPointer` must be injective on the domain of all subterms of the input. In the correctness theorem, it is convenient to express this condition as the existence of a function `f: Pointer → TermPointer`, corresponding to address lookup, which is left and right inverse to `GetPointer` on all subterms of the input.

Proving the correctness of the algorithm extends with moderate effort from the previous algorithm with some additional side lemmas and boilerplate. However, showing that the pointer assignment is correct is significantly more challenging. We define a function `add_pointer` assigning pointers with depth-first, preorder traversal of the syntactic tree, such as in Example 2.10.1.

Example 2.10.1. The following trees represent the formula $\neg(x \wedge \neg y)$ as respectively a `Term` and the corresponding `TermPointer`.



`Term` `TermPointer`

To construct the required inverse function, we first compute the list of subterms of a term, and map a pointer to the first term of the list with this pointer. It seems very obvious from the definition of our pointer assignment function `add_pointer` that there exists only one such term and hence that the two functions are inverses of each other, yet this was surprisingly difficult to prove, and required a lot of intermediate lemmas about the structure of subterms, the monotonicity of `add_pointer` along subterms, correspondence between the pointer and non-pointer version of terms, etc. The development is however independent of the orthologic proof system, and only depends upon the number and arity of constructors of orthologic terms. Hence, the theorems can straightforwardly be transferred to any other algebraic datatypes.

2.10.6 Proof-producing tactic

An alternative to verified decision procedures is *proof-producing decision procedures*, which also allows to implement tactics for proof assistants. Instead of relying on a soundness theorem, a proof-producing tactic will compute a proof (in Rocq, a proof term) that is then checked by the logical kernel.

Implementing a proof-producing version of the proof search algorithm in OCaml, we obtain a Rocq tactic, which we refer to as `olcert_goal`. In theory, this should be less efficient, as producing a proof takes additional time, but in practice an OCaml

implementation (leveraging, in particular, mutability) is a faster computation method than reducing lambda-terms. Compared to `solveOL_pointer`, it has the additional practical advantage of persistence of its memoization across calls, as the pointers are assigned once and memoization is maintained globally. Moreover, when possible, we memoize results as fields of the term nodes rather than in a map, making retrieval more efficient.

On the other hand, the proof term produced has size up to $\mathcal{O}(n^2)$, while the proof term of `solveOL_pointer` has constant size. Moreover, the proof-producing tactic is not verified, so it is not guaranteed that it does not contain a bug and outputs an incorrect proof. In fact, because Rocq's logic is rather complex and its inner workings hard to work with (and sparsely documented), it is quite unlikely that our implementation does not have issues with edge cases, for example involving universes or unification variables. This is less likely with reflection-based tactics, where only the reification step is sensible to such issues.

2.10.7 Normalization tactic and validity solver

Many formulas are not completely provable in the fragment of orthologic, in which case the decision procedure alone cannot do anything, even if most of the structure of the formula reduces under orthologic rules. The orthologic simplification algorithm (Theorem 2.4.13) on the other hand is widely applicable, and can make substantial progress on a goal even if the formula is not equivalent to true. This is also very useful if the propositional formulas are built on some theories, such as arithmetic, in which case even classical boolean decision procedures would not work. We implement this simplifier in OCaml as a Rocq plugin, using the above tactic to efficiently prove equivalence with the original expression. This yields a generic simplification tactic for terms of type `bool`. An OCaml algorithm computes the normal form f' of a formula f, and we then use either the proof-producing tactic or the reflection tactic to prove $f = f'$.

Finally, we can leverage this normalization tactic to implement a boolean solver which alternates between branching on a variable and normalizing the formulas, as in Lisa's `Tautology` solver (Subsection 3.4.3).

```
Theorem test_tauto02_0 (x0 x1: bool) :
  ! (((! x0 && ! x0) || (x0 && x1) || (x0 && ! x1) || (! x0 && ! x0)) &&
    ((x1 && x0) || (! x0 && x1) || (! x0 && ! x1) || (! x0 && x1)))
    =
  true.
Proof.
  olnormalize.
  (* (! x1 || ! x0) && (x0 || x1) && (x0 || ! x1)   = true*)
Qed.
```

This concludes our formalization. We formalized orthologic in the Rocq proof assistant, including soundness and completeness of OL^+ (Theorem 2.2.5, Theorem 2.2.6) and the cut elimination theorem for orthologic (Theorem 2.2.10). Doing so, we discovered a

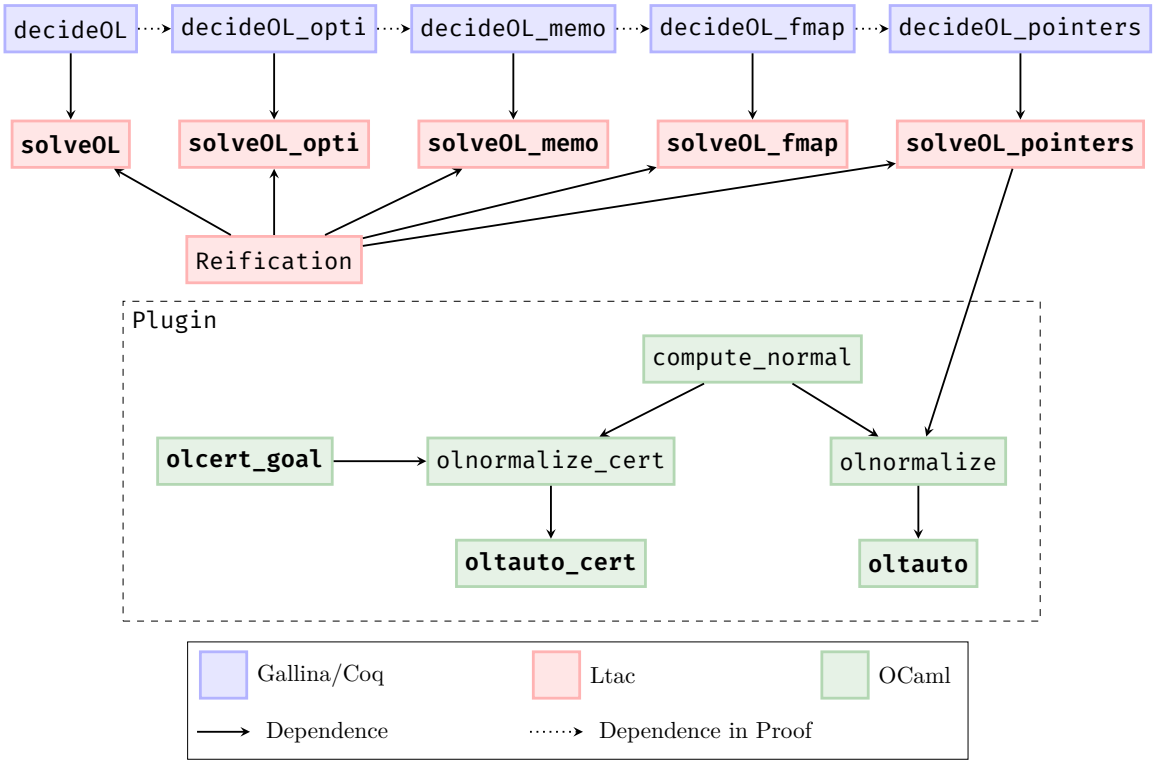


Figure 2.7: Diagram illustrating the interaction between each algorithm and tactic. Elements in bold have been benchmarked.

missing edge case in our original paper proof that was published in [40] (an erratum has been made available [38]), demonstrating the usefulness of mechanization. We implemented and verified a proof search procedure for orthologic, and used reflection to obtain a proof tactic that decides equalities and inequalities in ortholattices. We improved the algorithm with important but typically imperative programming features: memoization and pointer equality, improving the algorithm from exponential to quadratic. We implemented additional specialized tactics enabling orthologic-based reasoning for Rocq users, all of which are available in a Rocq plugin. Figure 2.7 presents an overview of the different components and their relations to each other. Evaluation of the various algorithms and corresponding tactics is presented in Subsection 2.12.2, where we show that each tactic meets its theoretical complexity.

2.11 Orthocomplemented bisemilattices

In this section we study the word problem and normalization problem for the algebraic class of orthocomplemented bisemilattices (*OCBSL*), which generalizes ortholattices further by disregarding the absorption law. *OCBSL* is weaker than the theory of ortholattices, but we will show that it admits an even faster normalization algorithm, running in time $\mathcal{O}(n \log(n)^2)$. Unlike reasoning in orthologic, which we based on proof systems, reasoning in *OCBSL* is based on a confluent term rewriting system. As a reminder, in lattices (including ortholattices), we can define an order relation $a \leq b$ by $a \wedge b = a$ or equivalently $a \vee b = b$. In bisemilattices, the two equations are not in general equivalent and yield two distinct order relations $\leq$ and $\sqsubseteq$ defined by $a \leq b \iff a \wedge b = a$ and $a \sqsubseteq b \iff a \sqcup b = b$. For this reason, we use the symbol $\sqcup$ for the join operation in bisemilattices instead of $\vee$.

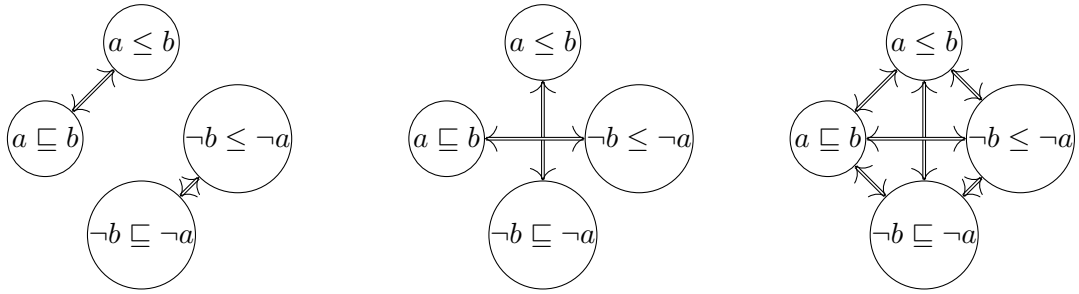
Definition 2.11.1. Orthocomplemented bisemilattices (or *OCBSL* for short) form an algebraic variety with signature $(\wedge, \sqcup, \neg, \bot, \top)$ satisfying laws L1-L8, L1'-L8' of Table 2.7. Orthocomplemented bisemilattices are dual to complemented lattices (which satisfy the absorption law but not the de Morgan law L8, L8') in the sense illustrated by Figure 2.8.

L1:	$x \sqcup y = y \sqcup x$	L1':	$x \wedge y = y \wedge x$
L2:	$x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$	L2':	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
L3:	$x \sqcup x = x$	L3':	$x \wedge x = x$
L4:	$x \sqcup \top = \top$	L4':	$x \wedge \bot = \bot$
L5:	$x \sqcup \bot = x$	L5':	$x \wedge \top = x$
L6:	$\neg\neg x = x$	L6':	same as L6
L7:	$x \sqcup \neg x = \top$	L7':	$x \wedge \neg x = \bot$
L8:	$\neg(x \sqcup y) = \neg x \wedge \neg y$	L8':	$\neg(x \wedge y) = \neg x \sqcup \neg y$

Table 2.7: Laws of algebraic structures with signature $(\wedge, \sqcup, \bot, \top, \neg)$. We call these structures orthocomplemented bisemilattices (*OCBSL*).

Equivalence up to commutativity of two formulas, say $a_1 \vee ... \vee a_n = b_1 \vee ... \vee b_n$ is easy to check by recursively checking if for every a_i there exists a b_j such that $a_i = b_j$, and conversely. This process is quadratic in the worst case, as we have to check equivalence of every possible pair of formulas $a_i = b_j$. This is essentially what our backward recursive algorithm for lattices and ortholattices does (see Subsection 2.3.3).

While decision procedures for lattices and ortholattices have quadratic complexity, we hope to obtain a faster algorithm for *OCBSL*. The first step is a more efficient approach to checking equivalence of two formulas modulo commutativity. Our algorithm thus uses as its starting point a variation of the algorithm of Aho, Hopcroft, and Ullman [52] for tree isomorphism, as it corresponds to deciding equality of two terms modulo commutativity in log-linear (i.e. $\mathcal{O}(n \cdot \log(n)^c)$) time. However, the theory we consider contains many more axioms than merely commutativity. The approach consists



(a) Complemented lattice(b) Orthocomplemented bisemilattice (c) Ortholattice

Figure 2.8: Bisemilattices satisfying absorption or de Morgan laws.

in finding an equivalent set of reduction rules, themselves understood modulo commutativity, that is suitable to compute a normal form of a given formula with respect to those axioms using the ideas of term rewriting [5]. Tree isomorphism serves two purposes: first, it helps to find application cases of reduction rules, and second, it compares the two terms of our word problem. In the final algorithm, both aspects are realized simultaneously.

2.11.1 Term rewriting systems

We briefly review basics of term rewriting systems. For a more complete treatment, see [5].

Definition 2.11.2. A **term rewriting system** is a set of rewriting rules of the form $e_l = e_r$ with the meaning that the occurrence of e_l in a term t can be replaced by e_r. e_l and e_r can contain free variables. To apply the rule, e_l is unified with a subterm of t, and that subterm is replaced by e_r with the same unifier. If applying a rewriting rule to t_1 yields t_2, we say that t_1 reduces to t_2 and write $t_1 \to t_2$. We denote by $\overset{*}{\to}$ the transitive closure of $\to$ and by $\overset{*}{\leftrightarrow}$ its transitive symmetric closure.

The set of identities characterizing a variety (such as L1-L8, L1'-L8' from Table 2.7) induces a term rewriting system, interpreting equalities from left to right. In that case $t_1 \overset{*}{\leftrightarrow} t_2$ coincides with the validity of the equality $t_1 = t_2$ in the theory given by the axioms [5, Theorem 3.1.12].

Definition 2.11.3. A term rewriting system is terminating if there exists no infinite chain of reducing terms $t_1 \to t_2 \to t_3 \to ....$

Fact 2.11.4. If there is a well-founded order $<$ (or, in particular, a measure m) on terms such that $t_1 \to t_2 \implies t_2 < t_1$ (or, in particular $m(t_2) < m(t_1)$) then the term rewriting system is terminating.

Definition 2.11.5. A term rewriting system is **confluent** if and only if: for all t_1, t_2, t_3, $t_1 \overset{*}{\to} t_2 \wedge t_1 \to^* t_3$ implies $\exists t_4 . t_2 \overset{*}{\to} t_4 \wedge t_3 \overset{*}{\to} t_4$.

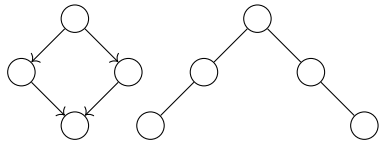


Figure 2.9: A DAG and the corresponding tree.

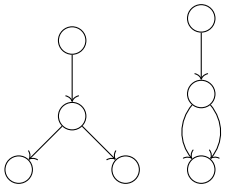


Figure 2.10: Two equivalent DAGs with different numbers of nodes.

Theorem 2.11.6 (Church-Rosser property)**.** [5, Chapter 2] A term rewriting system is confluent if and only if $\forall t_1, t_2.(t_1 \overset{*}{\leftrightarrow} t_2) \implies (\exists t_3.t_1 \overset{*}{\to} t_3 \wedge t_2 \overset{*}{\to} t_3)$.

A terminating and confluent term rewriting system directly implies decidability of the word problem for the underlying structure, as it makes it possible to compute the normal form of two terms to check if they are equivalent. Note that commutativity is not a terminating rewriting rule, but similar results hold if we consider the set of all terms, as well as rewrite rules, modulo commutativity [5, Chapter 11], [76]. To efficiently manipulate terms modulo commutativity and achieve log-linear time, we will employ an algorithm for comparing trees with unordered children.

2.11.2 Directed acyclic graph equivalence

The structure of formulas with commutative nodes corresponds to the usual mathematical definition of a labelled rooted tree, i.e. an acyclic graph with one distinguished vertex (root) where there is no order on the children of a node. For this reason, we use as our starting point the algorithm of Aho, Hopcroft, and Ullman for tree isomorphism [52, Page 84, Example 3.2].

As usual, we account for structure sharing by representing formulas as directed acyclic graphs. Any DAG can be transformed into a rooted tree by duplicating subgraphs corresponding to nodes with multiple parents, as in Figure 2.9. This transformation in general results in an exponential blow-up in the number of nodes. Dually, using DAGs instead of trees can exponentially shrink space needed to represent certain terms.

Checking for equality between *ordered* trees or DAGs is easy in linear time: we simply recursively check equality between the children of two nodes.

Definition 2.11.7. Two ordered nodes τ and π with children $\tau_0, \ldots, \tau_m$ and $\pi_0, \ldots, \pi_n$ are equivalent (denoted $\tau \sim \pi$) if and only if

$$label(\tau) = label(\pi),\ m = n \text{ and } \forall i \leq n, \tau_i \sim \pi_i$$

For unordered trees or DAGs, the equivalence checking is less trivial, as the naive algorithm has exponential complexity due to the need to find the adequate permutation.

Listing 2.19: Unordered DAG equivalence

```
1  import scala.collection.mutable.HashMap
2  case class Node(label: String, children: List[Node])
3
4  // Returns true iff two unordered DAGs `tau` and `pi` are equivalent.
5  def unorderedDagEquivalent(tau: Node, pi: Node): Boolean =
6    val dcodes = HashMap.empty[(String, List[Int]), Int]
7    val dmap   = HashMap.empty[Node, Int]
8
9    val sTau = reverseTopologicalOrder(tau)
10   val sPi  = reverseTopologicalOrder(pi)
11
12   for n ← (sTau ++ sPi) do
13     val ln   = n.children.map(ch ⇒ dmap(ch)) // codes of children
14     val rn   = (n.label, ln.sorted)          // canonical signature
15     val code = dcodes.getOrElseUpdate(rn, dcodes.size)
16     dmap.update(n, code)
17
18   dmap(tau) == dmap(pi)
```

Definition 2.11.8. Two unordered nodes τ and π with children $\tau_0, \ldots, \tau_m$ and $\pi_0, \ldots, \pi_n$ are equivalent (denoted $\tau \sim \pi$) if and only if

$$label(\tau) = label(\pi),\ m = n \text{ and there exists a permutation } p \text{ s.t. } \forall i \leq n, \tau_{p(i)} \sim \pi_i$$

For trees, note that this definition of equivalence corresponds exactly to isomorphism. It is known that DAG-isomorphism is GI-complete [100], so it is conjectured to have complexity greater than PTIME. Fortunately, this does not prevent our solution because our notion of equivalence on DAGs is not the same as isomorphism on DAGs. In particular, two DAGs can be equivalent without having the same number of nodes, i.e. without being isomorphic, as Figure 2.10 illustrates.

Listing 2.19 is the generalization of Aho, Hopcroft, and Ullman's algorithm. It decides in log-linear time if two labelled (unordered) DAGs are equivalent according to Definition 2.11.8. The algorithm generalizes straightforwardly to DAGs with a mix of ordered and unordered nodes: if a node is ordered, we skip the sorting operation in line 7.

The algorithm works bottom to top. We first sort the DAG in reverse topological order using, for example, Kahn's algorithm [54]. This way, we explore the DAG starting from a leaf and finishing with the root. It is guaranteed that when we treat a node, all its children have already been treated.

The algorithm recursively assigns codes to the nodes of both DAGs. In the unlabelled case:

- The first node, necessarily a leaf, is assigned the integer 0

- The second node gets assigned 0 if it is a leaf or 1 if it is a parent of the first node
- For any node, the algorithm makes a list of the integer assigned to that node's children and sort it (if the node is commutative). We call this the signature of the node. Then it checks if that list has already been seen. If yes, it assigns to the node the number that has been given to other nodes with the same signature. Otherwise, it assigns a new integer to that node and its signature.

Lemma 2.11.9 (Listing 2.19 correctness)**.** The codes assigned to any two nodes n and m of $s_\tau +\!\!+ s_\pi$ are equal if and only if $n \sim m$.

Proof. Let n and m denote any two DAG nodes. By induction on the height of n:

- In the case where n is a leaf, we have $r_n = (n.label, Nil)$. Note that for any node n, $dmap(n) = dcodes(r_n)$. Since every time the map $dcodes$ is updated, it is with a completely new number, $dcodes(r_n) = dcodes(r_m)$ if and only if $r_n = r_m$, i.e. if and only if $m.label = n.label$ and m has no children (like n).
- In the case where n has children n_i, again $dcodes(r_n) = dcodes(r_m)$ if and only if $r_m = r_n$, which is equivalent to ($m.label = n.label$ and $sort(l_m) = sort(l_n)$). This means there is a permutation of children of n such that $\forall i, dcodes(n_{p(i)}) = dcodes(m_i)$. By induction hypothesis, this is equivalent to $\forall i, n_{p(i)} \sim m_i$. Hence we find that $dmap(n) = dmap(m)$ if and only if both:

  1. Their labels are equal
  2. There exists a permutation p s.t. $n_{p(i)} \sim m_i$

  i.e. n and m have the same code if and only if $n \sim m$.

□

Corollary 2.11.10. The algorithm returns True if and only if $\tau \sim \pi$.

Time complexity Using Kahn's algorithm, sorting τ and π is done in linear time. Then the loop touches every node a single time. Inside the loop, the first line takes linear time with respect to the number of children of the node and the second line takes log-linear time with respect to the number of children. Since we use HashMaps, the last instructions take effectively constant time (because hash code is computed from the address of the node and not its content).

So for a general DAG, the algorithm runs in time at most log-quadratic in the number of nodes. Note however that for DAGs with bounded number of children per node as well as for DAGs with bounded number of parents per nodes, the algorithm is log-linear. In fact, the algorithm is log-linear with respect to the total number of edges in the graph. For this reason, the algorithm is still only log-linear in input size. It also follows that the algorithm is always at most log-linear with respect to the tree or formula underlying the DAG, which may be much larger than the DAG itself. Moreover, there exists cases

where the algorithm is log-linear in the number of nodes, but the underlying tree is exponentially larger. The full binary symmetric graph is such an example.

2.11.3 Word problem on orthocomplemented bisemilattices

We will use the previous algorithm for DAG equivalence to account for commutativity (axioms L1, L1'), but we need to combine it with the remaining axioms. From now on we work with axioms L1-L8, L1'-L8' from Table 2.7. To handle associativity (L2, L2'), we will work with a multiary version of $\sqcup$ (which corresponds to a multiary fan-in gate in a boolean circuit). The plan is to express the remaining axioms as reduction rules. Of rules L2-L8 and L2'-L8', all but L8 and L8' reduce the size of the term when applied from left to right, and hence seem suitable as rewrite rules.

It may seem that the simplest way to deal with de Morgan law is to use it (along with double negation elimination) to transform all terms into negation normal form. It happens, however, that doing this causes trouble when trying to detect application cases of rule L7 (complementation). Indeed, consider the following term:

$$f = (a \wedge b) \sqcup \neg(a \wedge b)$$

Using complementation it clearly reduces to $\top$, but pushing into negation-normal form, it would first be transformed to $(a \wedge b) \sqcup (\neg a \sqcup \neg b)$. To detect that these two disjuncts are actually opposite requires recursively verifying that $\neg(a \wedge b) = (\neg a \sqcup \neg b)$.

It is actually simpler to apply the de Morgan law the following way:

$$x \wedge y = \neg(\neg x \sqcup \neg y)$$

Instead of removing negations from the formula, we remove one of the binary semilattice operators. (Which one we keep is arbitrary; we chose to keep $\sqcup$.) Now, when we look if rule L7 can be applied to a disjunction node (i.e. two children y and z such that $y = \neg z$), there are two cases: if x is not itself a negation, i.e. it starts with $\sqcup$, we compute $\neg x$ code from the code of x in constant time. If there is some x' such that $x = \neg x'$ then $\neg x \sim x'$ so the code of $\neg x$ is simply the code of x', in constant time as well. Hence we obtain the code of all children and their negation and we can sort those codes to look for collisions, all of it in time linear in the number of children.

We now restate the axioms L1-L8, L1'-L8' in this updated language in Table 2.8.

It is straightforward and not surprising that axiom A8 as well as A1'-A8' all follow from axioms A1-A7, so A1-A7 are actually complete for our theory.

Confluence of the rewriting system

In our equivalence algorithm, A1 is taken care of by the arbitrary but consistent ordering of the nodes. Axioms A2-A7 form a term rewriting system. Since all those rules reduce the size of the term, the system is terminating in a number of steps linear in the size of

$A1:$	$\bigsqcup(\ldots, x_i, x_j, ...) = \bigsqcup(\ldots, x_j, x_i, ...)$	$A1':$	$\neg\bigsqcup(\neg x, \neg y) = \neg\bigsqcup(\neg y, \neg x)$
$A2:$	$\bigsqcup(\vec{x}, \bigsqcup(\vec{y})) = \bigsqcup(\vec{x}, \vec{y})$	$A2':$	$\neg\bigsqcup(\neg\vec{x}, \neg\neg\bigsqcup(\neg\vec{y})) = \neg\bigsqcup(\neg\vec{x}, \neg\vec{y})$
	$\bigsqcup(x) = x$		
$A3:$	$\bigsqcup(x, x, \vec{y}) = \bigsqcup(x, \vec{y})$	$A3':$	$\neg\bigsqcup(\neg x, \neg x, \neg\vec{y}) = \neg\bigsqcup(\neg x, \neg\vec{y})$
$A4:$	$\bigsqcup(\top, \vec{x}) = \top$	$A4':$	$\neg\bigsqcup(\neg\bot, \neg\vec{y}) = \bot$
$A5:$	$\bigsqcup(\bot, \vec{x}) = \bigsqcup(\vec{x})$	$A5':$	$\neg\bigsqcup(\neg\top, \neg\vec{x}) = \neg\bigsqcup(\neg\vec{x})$
$A6:$	$\neg\neg x = x$		
$A7:$	$\bigsqcup(x, \neg x, \vec{y}) = \top$	$A7':$	$\neg\bigsqcup(\neg x, \neg\neg x, \neg\vec{y}) = \bot$
$A8:$	$\neg\bigsqcup(x_1, ...x_i) = \neg\bigsqcup(\neg\neg x_1, ...\neg\neg x_i)$	$A8':$	$\neg\neg\bigsqcup(\neg x_1, ...\neg x_i) = \bigsqcup(\neg x_1, ...\neg x_i)$

Table 2.8: Laws of algebraic structures $(S, \sqcup, \bot, \top, \neg)$, equivalent to L1-L8, L1'-L8' under de Morgan transformation.

the term. We will next show that it is confluent. We will thus obtain the existence of a normal form for every term, and will finally show how our algorithm computes that normal form.

Definition 2.11.11. Consider a pair of reduction rules $l_0 \to r_0$ and $l_1 \to r_1$ with disjoint sets of free variables such that there exists some D and a fresh variable x and a non-variable term s such that $l_0 = D[x := s]$. Let σ be the most general unifier of s and l_1 (in particular, $\sigma s = \sigma l_1$). Then $(\sigma(r_0), \sigma(D)[x := \sigma r_1])$ is called a *critical pair*.

Informally, a critical pair is a most general pair of terms (with respect to unification) (t_1, t_2) such that for some t_0, $t_0 \to t_1$ and $t_0 \to t_2$ via two "overlapping" rules. They are found by matching the left-hand side of a rule with a non-variable subterm of the same or another rule.

Example 2.11.12 (Critical Pairs)**.**

1. Matching left-hand side of A6 with the subterm $\neg x$ of rule A7, we obtain the pair

$$(\top, \bigsqcup(\neg x, x, \vec{y}))$$

   which arises from reducing the term $t_0 = \bigsqcup(\neg x, \neg\neg x, \vec{y})$ in two different ways.

2. Matching left-hand sides of A2 and A7 gives

$$(\bigsqcup(\vec{x}, \vec{y}, \neg\bigsqcup(\vec{y})), \top)$$

   which arises from reducing $\bigsqcup(\vec{x}, \bigsqcup(\vec{y}), \neg\bigsqcup(\vec{y}))$ using A2 or A7.

3. Matching left-hand sides of A5 and A7 gives

$$(\neg\bot, \top)$$

   which arise from reducing $\bot \sqcup \neg\bot$ in two different ways.

$$
\begin{array}{rl}
A1: & \bigsqcup(\ldots, x_i, x_j, ...) = \bigsqcup(\ldots, x_j, x_i, ...) \\
A2: & \bigsqcup(\vec{x}, \bigsqcup(\vec{y})) = \bigsqcup(\vec{x}, \vec{y}) \\
A2b: & \bigsqcup(x) = x \\
A3: & \bigsqcup(x, x, \vec{y}) = \bigsqcup(x, \vec{y}) \\
A4: & \bigsqcup(\top, \vec{x}) = \top \\
A5: & \bigsqcup(\bot, \vec{x}) = \bigsqcup(\vec{x}) \\
A6: & \neg\neg x = x \\
A7: & \bigsqcup(x, \neg x, \vec{y}) = \top \\
A9: & \bigsqcup(\vec{x}, \vec{y}, \neg \bigsqcup(\vec{y})) = \top \\
A10: & \neg\bot = \top \\
A11: & \neg\top = \bot
\end{array}
$$

Table 2.9: Terminating and confluent (up to commutativity) set of rewrite rules equivalent to L1-L8, L1'-L8'

Theorem 2.11.13 ([5, Chapter 6])**.** A terminating term rewriting system is confluent if and only if all critical pairs (t_1, t_2) are joinable i.e. $\exists t_3.\ t_1 \xrightarrow{*} t_3 \wedge t_2 \xrightarrow{*} t_3$.

In the first of the previous examples, the pair is clearly joinable by commutativity and a single application of rule A7 itself. The second example is more interesting. Observe that $\bigsqcup(\vec{x}, \vec{y}, \neg \bigsqcup(\vec{y})) = \top$ is a consequence of our axiom, but the left part cannot be reduced to $\top$ in general in our system. To solve this problem we need to add the rule A9: $\bigsqcup(\vec{x}, \vec{y}, \neg \bigsqcup(\vec{y})) = \top$. Similarly, the third example forces us to add A10: $\neg\bot = \top$ to our set of rules. From A10 and A6 we then derive the expected rule A11: $\neg\top = \bot$.

Complete terminating confluent rewrite system

The analysis of all possible pairs of rules to find all critical pairs is straightforward. It turns out that A9, A10 and A11 are the only rules we need to add to our system to obtain confluence. All those pairs are joinable, i.e. reduce to the same term, which implies, by Theorem 2.11.13, that the system is confluent. Table 2.9 shows the complete set of reduction rules (as well as commutativity).

Since the system A2-A11 considered over the language $(S, \bigsqcup, \neg, \bot, \top)$ modulo commutativity of $\bigsqcup$ is terminating and confluent, it implies the existence of a normal form function (as in Definition 2.4.2). For any term t, we note its normal form $t{\downarrow}$. In particular, for any two terms t_1 and t_2, we have $t_1 \sim_{OCBSL} t_2 \iff t_1 \overset{*}{\leftrightarrow} t_2 \iff t_1{\downarrow} = t_2{\downarrow}$.

2.11.4 Algorithm and complexity

The rewriting system readily gives us a quadratic algorithm. Indeed, using our base algorithm for DAG equivalence, we can check, in linear time, for application cases of any one of rewriting rules A2-A11 of Table 2.9 modulo commutativity. Since a term can only be reduced up to n times, the total time spent before finding the normal form of a

term is at most quadratic. It is however possible to find the normal form of a term in a single pass of our equivalence algorithm, resulting in a more efficient algorithm.

Combining rewrite rules and tree isomorphism

We give an overview on how to combine rules A2-A7, A9, A10, A11 within the tree isomorphism algorithm, which is presented in Listing 2.20. For conciseness, we omit the dynamic programming optimizations allowed by structure sharing in DAGs (which would store the normal form and additionally check if a node was already processed.) For each rule, we indicate the most relevant lines of the algorithm.

A2 (Associativity) When analysing a $\bigsqcup$ node, after the recursive call, find all children that are $\bigsqcup$ themselves and replace them by their own children. This is simple enough to implement, but there is actually a caveat with this in terms of complexity. We will come back to this in subsubsection 2.11.4.

A3 (Idempotence) This follows from the fact that we eliminate duplicate children in disjunctions. When reaching a $\bigsqcup$ node, after having sorted the code of its children, remove all duplicates before computing its own code.

A4, A5 (Bounds) To account for those axioms, we reserve a special code for the nodes $\top$ and $\bot$. For A4, when we reach some $\bigsqcup$ node, if it has $\top$ as one of its children, we accordingly replace the whole node by $\top$. For A5, we just remove nodes with the same codes as $\bot$ from the parent node before computing its own code.

A6 (Involution) When reaching a negation node, if its child is itself a negation node, replace the parent node by its grandchild before assigning it a code.

A7 (Complement) As explained earlier, our representation of nodes let us do the following to detect cases of A7: First remember that we already applied double negation elimination, so that two "opposite" nodes cannot both start with a negation. Then we can simply separate the children between negated and non-negated (after the recursive call), sort them using their assigned code and look for collisions.

A9 (Also Complement) This rule is slightly more tricky to apply. When analysing a $\bigsqcup$ node x, after computing the code of all children of x, find all children of the form $\neg\bigsqcup$. For every such node, take the set of its own children and verify if it is a subset of the set of all children of x. If yes, then rule A9 applies. Said otherwise, we look for collisions between grandchildren (through a negation) and children of every $\bigsqcup$ node.

A10, A11 (Identities) These rules are simple. In a $\neg$ node, if its child has the same code as $\bot$ (resp $\top$), assign code $\top$ (resp $\bot$) to the negated node.

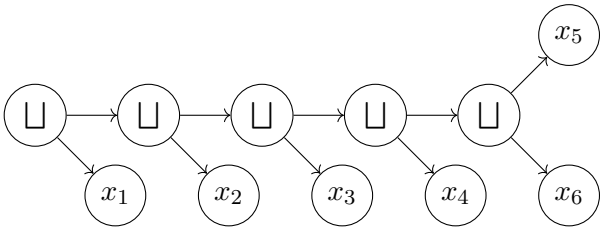


Figure 2.11: A term with quadratic runtime.

Case of quadratic runtime for the basic algorithm

All the rules we introduced in the previous section into Listing 2.19 take time, when applied to a node, (log)linear in the number of children of that node. This is not more than the time we spent in the DAG/tree isomorphism algorithm. For A3, checking for duplicates is done in linear time in an ordered data structure. A4 and A5 (Bounds) consist in searching for specific values, which take logarithmic time in the size of the list. A6 (Involution) takes constant time. A7 (Complement) is detected by finding a collision between two separate ordered lists, also easily done in (log) linear time. A9 (Also complement) consists in verifying if grandchildren of a node are also children, and since children are sorted this takes log-linear time in the number of grandchildren. A10 and A11 take constant time. Hence, the total time complexity is $\mathcal{O}(n \log(n))$, as in the algorithm for tree isomorphism.

As stated in Subsection 2.11.2 regarding the algorithm for DAG equivalence whose complexity we aim to preserve, the time complexity analysis crucially relies on the number of parent-child pairs. However, this is generally not true in the presence of associativity. Indeed, consider the term represented in Figure 2.11. The 5th $\bigsqcup$ has 2 children, but after applying A2, the 4th has 3 children, the 3rd has 4 children and so on. On the generalization of such an example, since an x_i is the child of all higher $\bigsqcup$ throughout the algorithm, its runtime would be quadratic. Of course, such a simple counterexample is easily solved by applying a leading pass of associativity reduction before actually running the whole algorithm. It turns out however that it is not sufficient, since cases of associativity can appear after the application of the other A-rules.

In fact, there is only one rule that can create cases of rule A2, and this rule is A6 (Involution). The remaining rules whose right-hand side can start with a $\bigsqcup$ have their left-hand side already starting with $\bigsqcup$. It may seem simple enough to also apply double negation elimination in a leading pass, but unfortunately, cases of A6 can also be created from other rules. It is easy to see, for similar reasons, that only the application of A2b ($\bigsqcup(x) = x$) can create such cases. And unfortunately, such cases of A2b can arise from rules A3 and A5 which can only be detected using the full algorithm. To summarize, the typical problematic case is depicted in Figure 2.12. This term is clearly equivalent to $\bigsqcup(x_1, x_2, x_3, x_4)$, but to detect it we must first find that z_1 and z_2 are equivalent to $\bot$, so we cannot simply solve it with an early pass.

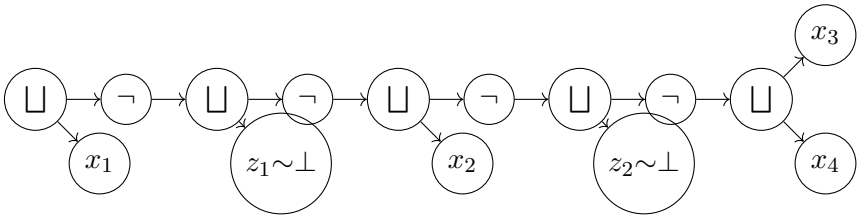


Figure 2.12: A non-trivial term with quadratic runtime.

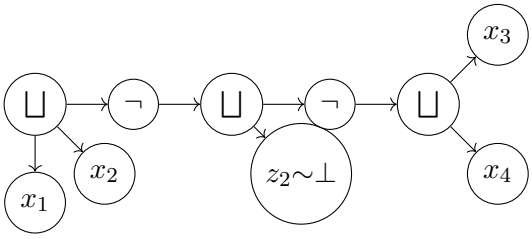


Figure 2.13: The term of Figure 2.12 during the algorithm's execution.

Final log-linear time algorithm

Fortunately, we can solve this problem at a logarithmic-only price. Observe that if we are able to detect early nodes which would cancel to $\bot$, the problem would not exist: when analysing a node, we would first call the algorithm on all subnodes equivalent to $\bot$, remove them and then when there is a single child left, remove the trivial disjunct, the double negation and the successive disjunction (as in Figure 2.12) before doing the recursive call on the unique non-trivial child. However, we of course cannot know in advance which child will be equivalent to $\bot$.

Moreover note (still using Figure 2.12) that if the z-child is as large as the non-trivial node, then even if we do the "useless" work, we at least obtain that the size of a tree is divided by two, and hence the potential depth of the tree as well. By standard complexity analysis, the time penalty would only be a logarithmic factor.

The previous analysis suggests the following solution, reflected in Listing 2.20 lines 28-29. When analysing a node, make recursive calls on children in order of their size, starting with the smallest up to the second biggest. If any of those children are non-zero, proceed as normal. If all (but possibly the last) children are equivalent to zero, then replace the current node by its biggest (and at this point not analysed) child, i.e. apply the second half of rule A2 (associativity). If applicable, apply double negation elimination and associativity as well before continuing the recursive call.

We illustrate this on the example of Figure 2.12. Consider the algorithm when reaching the second $\bigsqcup$ node. There are two cases:

1. Suppose z_1 is a smaller tree than the non-trivial child. In this case the algorithm will compute a code for z_1, find that it is $\bot$ and delete it. Then the non-trivial node is a single child, so the whole disjunction is removed. Hence, the double

negation can be removed and the two consecutive disjunctions of x_1 and x_2 merged, obtaining the term illustrated in Figure 2.13. In particular, we did not compute a code for the two deleted $\bigsqcup$ nodes, which is exactly what we wanted for our initial analysis.

2. Suppose z_1 is a larger tree than the non-trivial child. In this case, we would first recursively compute the code of the non-trivial child and then detect that $z_1 \sim_{OCBSL} \bot$. We indeed computed the code of the disjunction that contains x_2 when it was unnecessary since we apply associativity anyway. This "useless" work consists in sorting and applying axioms to the true children of the node (in this case x_2, x_3 and x_4) and takes time log-linear in the number of such children. In particular, it is bounded by the size of the subtree itself and we know it is the smallest of the two.

An analogous situation can arise from the use of rule A3 (idempotence), but trivially, the two subtrees must have the same number of (real) subnodes, so that the same reasoning holds.

Denote by $|t|$ the size of a node, i.e. the number of descendants of t. We compute the penalty of useless work we incur by computing children of a node t in the wrong order, i.e. by computing a non-$\bot$ child t_w when all others are $\bot$. t_w cannot be the largest child of t for otherwise we would have found that all other children are $\bot$ before needing to compute t_w. Hence $|t_w| \leq |t|/2$. It follows that the total amount of useless work is bounded by $\log(|t|) \cdot W(t)$, where

$$W(t) \leq |t|/2 + \sum_i W(t_i) \quad \text{for } \sum_i |t_i| < |t|.$$

It is clear that $W(t)$ is maximized when t has exactly two children of equal size:

$$W(t) \leq |t|/2 + 2 \cdot W(t/2)$$

By observing that we can divide $|t|$ by 2 only $\log(|t|)$ times,

$$W(t) \leq \sum_{m=1}^{\log(t)} 2^m \cdot |t|/2^m$$

so we obtain $W(t) = \mathcal{O}(|t| \log(|t|))$ and hence the total runtime is $\mathcal{O}(|t|(\log |t|)^2)$.

Listing 2.20: Algorithm for equivalence of formulas in *OCBSL*

```
def equivalentTrees(tau: Term, pi: Term): Boolean =
  val codesSig: HashMap[(String, List[Int]), Int] = HashMap.empty
  codesSig.update(("zero", Nil), 0); codesSig.update(("one", Nil), )
  val codesNodes: HashMap[Term, Int] = HashMap.empty
  def updateCodes(sig: (String, List[Int]), n: Term): Unit =  ...
  def bool2const(b:Boolean): String = if b then "one" else "zero"
  def rootCode(n: Term): Int =
    val L = pDisj(n, Nil).map(codesNodes).sorted.filter(_ ≠ 0).distinct
    if L.isEmpty          then updateCodes(("zero", Nil), n)
    else if L.length ==  then codesNodes.update(n, L.head)
    else if L.contains() || checkForContradiction(L) then
      updateCodes(("one", Nil), n)
    else updateCodes(("or", L), n)
    codesNodes(n)
  def pDisj(n:Term, acc:List[Term]): List[Term] = n match
    case Variable(id) ⇒
      updateCodes((id.toString, Nil), n); return n :: acc
    case Bot ⇒ ...
    case Negation(child) ⇒ pNeg(child, n, acc)
    case Disjunction(children) ⇒ children.foldLeft(acc)(pDisj)
  // under negation
  def pNeg(n:Term, parent:Term, acc:List[Term]): List[Term] = n match
    case Negation(child) ⇒ pDisj(child, acc)
    case Variable(id) ⇒
      updateCodes((id.toString, Nil), n)
      updateCodes(("neg", List(codesNodes(n))), parent)
      parent :: acc
    case Bot ⇒ ...
    case Disjunction(children) ⇒
      val r0 = orderBySize(children)
      val r = r0.tail.foldLeft(Nil)(pDisj)
      val r2 = r.map(codesNodes).sorted.filter(_ ≠ 0).distinct
      if isEmpty(r2) then pNeg(r0.head, parent, acc)
      else
        val s = pDisj(r0.head, r)
        val s2 = s zip (s map codesNodes)
        val s3 = s2.sorted.filter(_ ≠ 0).distinct // all wrt. 2nd element
        if s3.contains() || checkForContradiction(s3)
        then
          updateCodes(("one", Nil), n); updateCodes(("zero", Nil), parent)
          parent :: acc
        else if isEmpty(s3) then
          updateCodes(("zero", Nil), n); updateCodes(("one", Nil), parent)
          parent :: acc
        else if s3.length ==  then pNeg(s3.head._, parent, acc)
        else updateCodes(("or", s3 map (_._2)), n)
          updateCodes(("neg", List(n)), parent)
          parent :: acc
    rootCode(tau) == rootCode(pi)
```

2.12 Experimental evaluation

In this section, we experimentally evaluate our normalization algorithms for *OL* and *OCBSL* presented in Section 2.4, as well as the *OL* proof search tactics that we have implemented in Rocq (Section 2.10).

Acknowledgement of contributions The implementation and evaluation of *OL* and *OCBSL* normalization in Stainless was carried out by Mario Bucev (Subsection 2.12.1). The implementation and evaluation of *OL* proof search tactics in Rocq was carried out in collaboration with Clément Pit-Claudel (Subsection 2.12.2).

2.12.1 Normalization

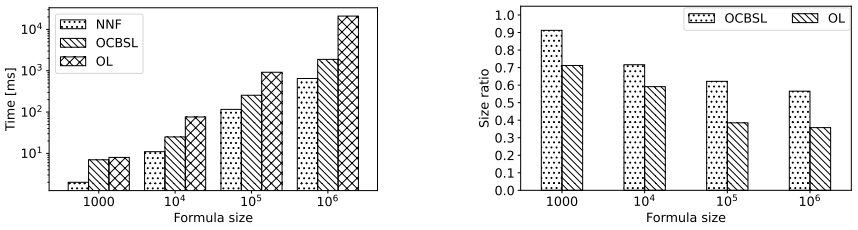
Our normalization algorithms for *OCBSL* and *OL* are, in theory, efficient: the *OCBSL* algorithm runs in time $\mathcal{O}(n \log(n)^2)$ and the *OL* algorithm in $\mathcal{O}(n^2)$. However, practical questions remain: Are they as efficient in practice as the theory suggests? Do they yield significant formula size reductions on realistic formulas? Do they help in real-world applications?

Both algorithms operate on DAG representations of propositional formulas, that is, *Boolean circuits*. Crucially, this representation is exponentially more efficient than tree representations in the worst case, as subformulas can be reused many times. Our implementations in Scala are available at *https://github.com/epfl-lara/lattices-algorithms*. They are reasonably optimized, using a form of hash-consing to represent formulas and efficient data structures for sets and maps, but do not contain true low-level optimizations which for SAT solvers represent orders of magnitude improvements.

We study two primary applications of normalization in verification: as a preprocessing technique that simplifies a formula before a solver is invoked, and as a canonicalization technique for persistent caching of previously proven verification conditions. Concretely, our experimental evaluation comprises three parts. First, we analyse the behaviour of the *OL* and *OCBSL* algorithms on large random formulas, to understand the feasibility of using them for normalization. Second, and most importantly, we show their impact through a new simplifier for verification conditions of the Stainless [45] verifier. The goal of the simplifier is to avoid the need to invoke a solver for some of the formulas by reducing them to `true`, as well as to normalize them before storing them in a persistent cache file. The cache avoids the need to repeatedly prove previously proven verification conditions. By using a more powerful normal form function, we improve the cache hit rate.

Randomly generated propositional formulas We first evaluate the two algorithms on randomly generated formulas. We measure the running time and the reduction in formula size. The random formulas are generated as follows.

Definition 2.12.1. A random formula is parameterized by a size s and a set of available



(a) Running time. (b) Size ratio, showing reduction.

Figure 2.14: (a) Median running time of NNF and the two algorithms (log-log scale). (b) Median size of the normalized formulas relative to the original formulas in NNF. $|X| = 50$ variables.

variables $X = \{x_1, \ldots, x_n\}$. Given a size s, if $s \leq 1$ then pick a variable uniformly at random from X or its negation and return it. Otherwise, pick t such that $0 < t < s$ and generate two formulas ϕ_1 and ϕ_2 of sizes t and $s - 1 - t$. Return uniformly at random $\text{And}(\phi_1, \phi_2)$ or $\text{Or}(\phi_1, \phi_2)$.

We show in Figure 2.14a the running time of both algorithms for various sizes of formulas. We ran the experiment 21 times for each formula size category and took the median. For comparison with a theoretically linear-time process, we also give the running time of the corresponding negation normal form transformation.

For a fairer comparison of the size reduction, we apply a basic simplification (flattening and transformation into negation normal form) to random formulas before computing their size. We compare the number of connectives before and after the simplification for both algorithms. We show the relative improvements of the *OL* and *OCBSL* algorithms compared to the original formulas for various sizes of formulas and 50 variables. We ran both algorithms 21 times and report the median results in Figure 2.14b.

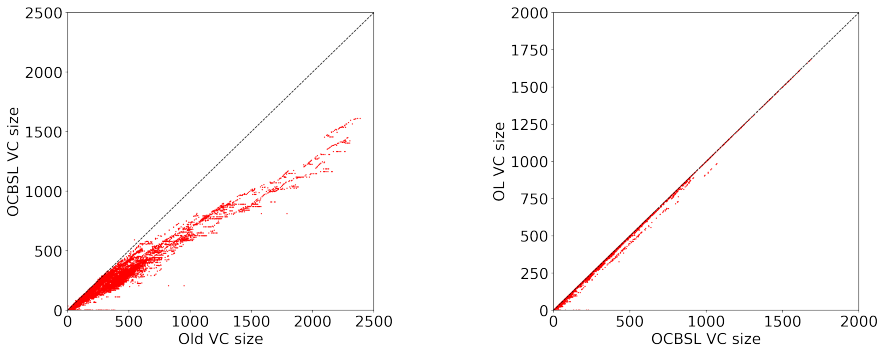
It is interesting to note that the *OL* normal form is consistently and significantly smaller than the *OCBSL* normal form, i.e., the absorption law (V4, V4', Table 2.1) actually allows non-trivial reductions in size. This confirms that, in general, there is a trade-off between speed and simplification strength for the two algorithms.

Caching verification conditions in Stainless We then modify the Stainless verifier [45][7], a publicly available tool for building formally verified Scala programs.

Our implementation adds two new simplifiers to Stainless: *OCBSL*-backed and *OL*-backed. They are part of Stainless release v0.9.8[8] and can be selected via the command line options `--simplifier=ocbsl` and `--simplifier=ol` respectively. For the *OL* simplifier, we have extended the $\leq_{OL}$ relation with 12 simple arithmetic and array rules.

We experimentally compare the two new simplifiers to the existing one (which we denote Old). We use two groups of benchmarks: (1) six Stainless case studies from the

[7] *https://github.com/epfl-lara/stainless/*
[8] *https://github.com/epfl-lara/stainless/releases/tag/v0.9.8*



(a) Old vs OCBSL. (b) OCBSL vs OL.

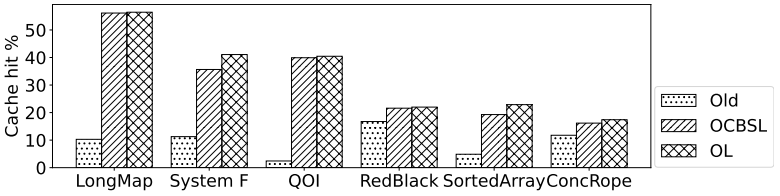
Figure 2.15: VCs (tree) size scatter plot from all benchmarks for Old, OCBSL and OL.

Bolts repository[9] that take a significant amount of time to verify, and (2) nine benchmark sets from automated grading of student assignments. Together, this constitutes around 84 000 lines of Scala code, specifications, and auxiliary assertions. We report the following metrics: the size of the VCs after simplification, the number of cache hits, the number of VCs simplified to `true`, the wall-clock time and the cumulative solving time. The wall-clock time comprises the full Stainless pipeline, from parsing the program to outputting the result, going through solver calls and VC simplification.

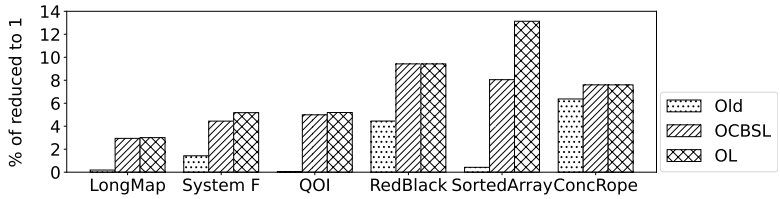
Evaluation on Bolts case studies We consider the following case studies from the mentioned Bolts repository:

- `LongMap` (9613 VCs, 7091 LOC), a mutable hash map, 64-bit integer keys, open addressing, formalized by Samuel Chassot (EPFL) and proven to behave equivalently to a list of (key, value) pairs.
- A type checker for `System F` (5040 VCs, 2501 LOC) formalized in Stainless by Andrea Gilot and Noé De Santo (EPFL). Among the key properties proven are type judgment uniqueness, preservation and progress.
- `QOI` (4487 VCs, 2812 LOC), an implementation of the Quite OK Image format. Decoding an encoded image is shown to yield the original image [15].
- `RedBlack`, a red-black tree (764 VCs, 796 LOC).
- `SortedArray` (472 VCs, 429 LOC), a mutable array preserving order on insertion. Developed for use in a simplified model of part of a file system [46].
- `ConcRope` (408 VCs, 621 LOC), a Conc-Tree rope [78], supporting amortized constant time append and prepend operations, based on a Leon formalization [60].

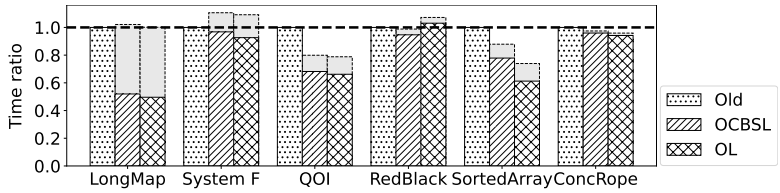
[9] *https://github.com/epfl-lara/bolts*



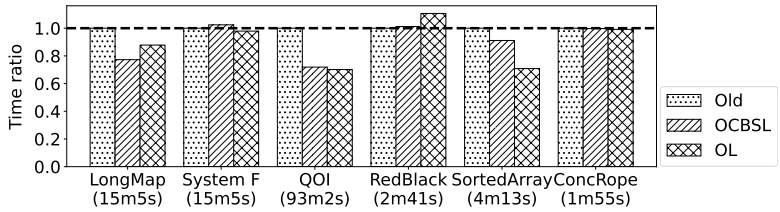
(a) Cache hits in a single run.



(b) VCs simplified to 1.



(c) Cumulative solving time.



(d) Wall-clock time.

Figure 2.16: Old, OCBSL and OL results for cache hits, VCs reduced to 1, solving and running time. (c), (d) are normalized with respect to Old. In (c), the gray boxes represent the time spared due to extra cache hits and VCs reduced to 1 compared to Old.

Figure 2.15 reports the size measurement, where we aggregate the results from all benchmarks. Figure 2.15a reveals some VCs with an increased size. Inspection of these VCs shows that the reason is that the new simplifiers always inline "simple expressions", such as field projection on free variables, instead of having them bound. On average, *OCBSL* decreases the size of the VCs by 37% compared to Old. *OL* reduces the size of the VCs slightly compared to *OCBSL* (Figure 2.15b).

In Figure 2.16a, we report the cache hit ratio. For the new simplifiers, reducing the formula size has the desired effect of noticeably increasing the hit ratio, especially for 4 out of 6 benchmarks. The additional power of *OL* helps for `System F` and `SortedArray`.

Figure 2.16c reports the solving time for the two simplifiers (normalized with respect to Old), as well as the solving time saved thanks to additional cache hits and VCs simplified to `true`. `ConcRope` and `RedBlack` do not benefit from the new simplifiers, while the other benchmarks do in various degrees. For `LongMap`, we observe that the solver did not benefit from the new simplifiers for non-cached VCs, but that caching improved total running time by a factor of 2. The `System F` benchmark shows a ratio exceeding 1, meaning that *OCBSL* and *OL* did not help the solver more than the extra time they took to run. For `QOI` and `SortedArray`, the combined ratio is less than 1: the new simplifiers helped the solver for non-cached VCs. *OL* performs significantly better than *OCBSL* in the `SortedArray` benchmark, thanks to the extension of the $\leq_{OL}$ relation with array rules.

Turning our attention to Figure 2.16d, we note that the time saved on solver calls is essentially offset by the extra work performed by the new simplifiers on three of the benchmarks. Moreover, `LongMap`, `SortedArray` and especially `QOI` have a net benefit over Old.

OCBSL and *OL* simplifiers show the greatest improvement on large VCs. Note that the outcome of a Stainless run highly depends on user-provided assertions, which were hand-tuned under the Old simplifier. It is thus possible that the new simplifiers have a disadvantage because they were not used during the verification process. The additional power provided by the new simplifiers may make writing such intermediate assertions easier and faster, so we expect the full advantage of the new simplifiers in newly developed verified software.

2.12.2 *OL* proof search tactics in Rocq

We now present the experimental evaluation of the various *OL* proof search algorithms implemented in Rocq, as described in Section 2.10. These implementations yielded six corresponding proof tactics for deciding equality in orthologic, each with a different time complexity[10]:

- `solveOL` (`"OL"`, $\mathcal{O}(2^n)$)

[10]Exact complexity of algorithms up to logarithmic factors depends on the precise model of computation; for simplicity, the algorithmic complexity below assumes the usual Word RAM model, where checking equality of words takes constant time, even though this does not precisely match Rocq's evaluation of terms.

solver	mean	std
btauto	35.96	13.30
OL+o+l	19.89	0.26
OL+o+m	0.36	0.01
OL+o+m+φ	0.25	0.01
OCaml	0.12	0.00

Table 2.10: Wall clock time required to prove an equality involving 30 variables.

- `solve_OL_opti` (`"OL+o"`, $\mathcal{O}(2^n)$)
- `solve_OL_memo` (`"OL+o+l"`, $\widetilde{\mathcal{O}}(n^5)$)
- `solve_OL_fmap` (`"OL+o+m"`, $\widetilde{\mathcal{O}}(n^3)$)
- `solve_OL_pointers` (`"OL+o+m+φ"`, $\widetilde{\mathcal{O}}(n^2)$)
- `olcert_goal` (`"OCaml"`, $\mathcal{O}(n^2)$)

The difference of complexity is not only theoretical, but also highly observable in practice. Consider the family of propositions (the same as those we used to demonstrate inefficiency in Scala, TypeScript and Flow in Subsection 2.9.6):

$$(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge ... \wedge (x_{n-1} \vee x_n)$$

$$= (x_2 \vee x_1) \wedge (x_4 \vee x_3) \wedge ... \wedge (x_n \vee x_{n-1})$$

These equalities hold by the laws of ortholattices. Table 2.10 shows the time taken by each implementation for $n = 30$. For comparison, we also include the runtime of `btauto`, the solver for boolean equalities included in Rocq's standard distribution.

To confirm the scaling characteristics of our implementation variants, we solved the family of equalities above for sizes ranging from 2 to 100 variables. Figure 2.17 shows our results. We confirm the theoretical complexity by verifying for each tactic that a polynomial of the right degree fits the measurements.

The proof-producing implementation in OCaml outperforms the implementation as a Rocq function (i.e. in Gallina), but it produces longer proof terms, and since its soundness is not proven it may contain bugs. But it also demonstrates the inadequacy of algorithms implemented directly in Rocq for practical purposes, as reduction of lambda-terms is not an efficient computational strategy. This suggests that in an ideal world, it should be possible to verify functions implemented in a general-purpose programming language and then trust their output, so that they can run as fast as possible. This was done in some flavour with the Candle proof assistant (a verified implementation of HOL Light) in [1].

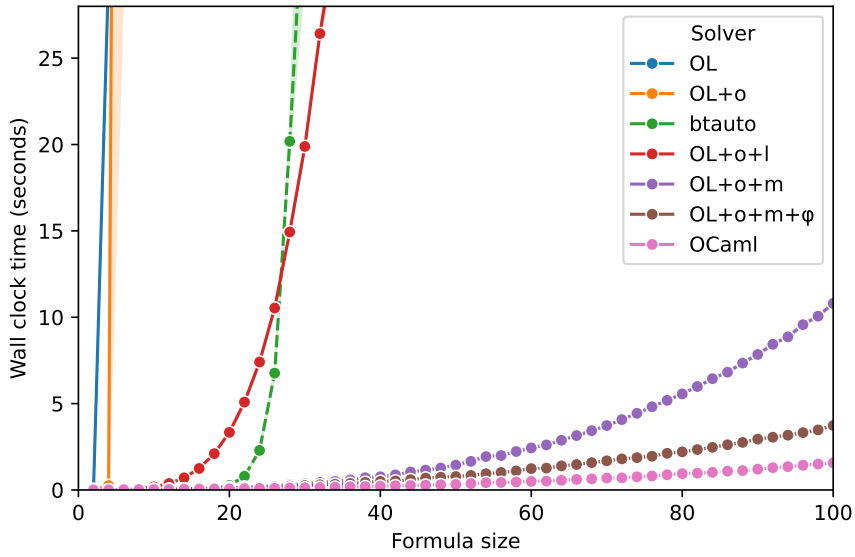


Figure 2.17: Wall clock time required to prove a family of equalities with sizes ranging from 2 to 100 variables. Shaded regions indicate 95% confidence intervals. Colours indicate which implementation was used. Benchmarks were run on an Intel Core i7-1365U CPU with 32GB RAM.

Evaluating propositional solvers To judge the tactics `oltauto` and `oltauto_cert` compared to the standard `btauto` solver, we generate a series of formulas according to the techniques of [70]. These formulas have a fixed structure and are believed to be typically hard for SAT solvers. Some of the formulas we generated are valid and some are not, but this does not affect the runtime meaningfully as all three algorithms do not stop early if a falsifying assignment is found and still explore the entire search space. We generate a total of 80 random formulas, with up to 20 different variables and 192 literals, and a timeout of 30 seconds. The results appear in Figure 2.18 and show that `oltauto_cert` (OCaml+n) consistently outperforms `oltauto` (OLT), which in turn consistently outperforms `btauto`, which scales very poorly with respect to formula size even with a low number of variables (< 6).

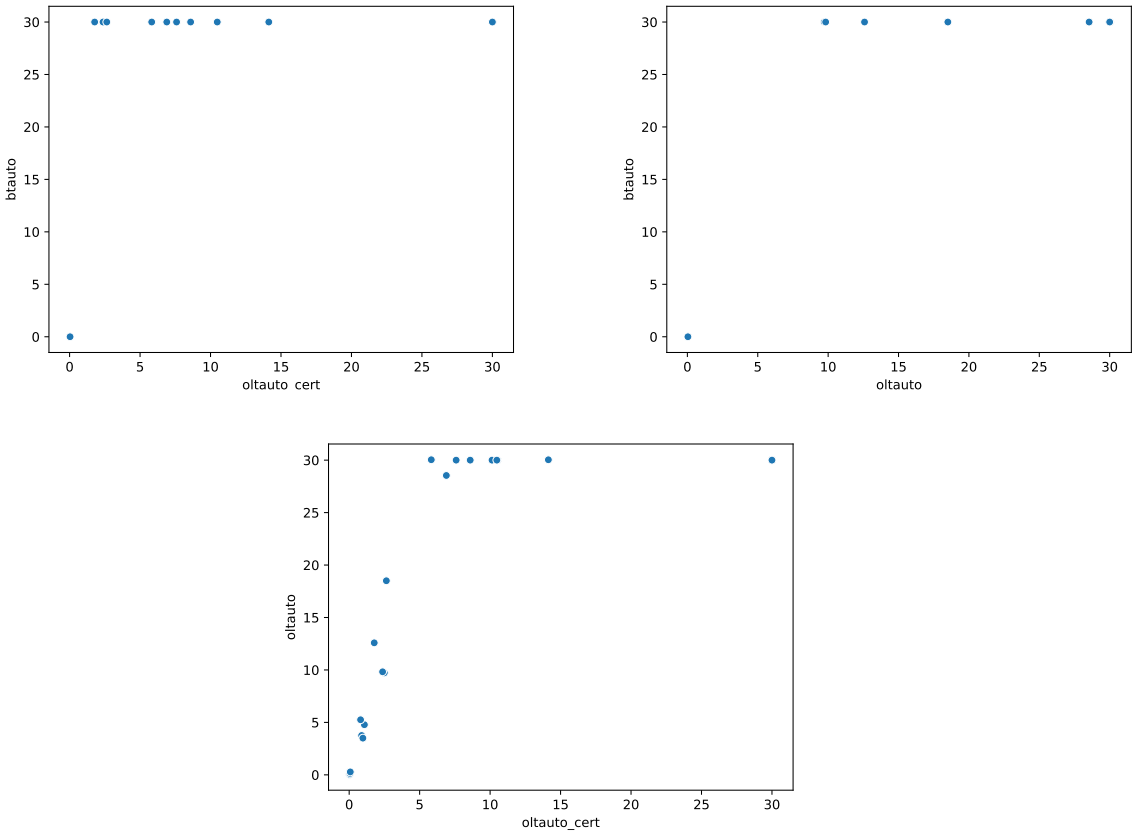


Figure 2.18: Wall clock time required to prove random hard non-clausal formulas. Each point is an experiment and indicates the running time for two of the three solvers.

3 Lisa

In this chapter, we present the theoretical foundations, design and library of the proof assistant Lisa. Lisa aims to use classical mainstream foundations of mathematics with first-order logic and set theory. Its design is inspired by the LCF line of proof assistants, including HOL Light [47], HOL4 [84] and Isabelle [97]. Its main axiomatic foundation, based on ZFC set theory, is closer to that of Mizar [69].

Lisa is built around a kernel, a minimalist trusted core that enables writing and verification of proofs. It provides a small set of primitive operations for constructing proofs that can be combined to form more complex proof strategies. Above the kernel, Lisa contains a more expressive high-level interface, proof automation algorithms named *tactics* and a library of mathematics made of definitions and theorems. This design allows for a clear separation between the trusted core of the system, whose main purpose is to guarantee with high confidence that only actually true theorems are accepted, and the rest of the system, which cannot compromise the soundness of the system even if it contains bugs, allowing it to be more complex and to offer more features.

As we explained in the introduction, Lisa is built on six design principles for proof systems: trust, efficiency, predictability, usability, interoperability and programmability. *Programmability* is a key focus of Lisa compared to most older proof assistants. Some have a dedicated proof script language but with limited automation features (e.g., Rocq, Isabelle), while in others proofs are written directly within the host language, which is better for automation but is typically hard to read and write (e.g., the HOL family). A unique feature of Lisa is that it offers the best of both worlds: all the expressiveness and features of a modern programming language, and an expressive and natural proof script language. This is possible because Lisa has a single unified implementation, proof writing and tactic language. Concretely, Lisa is implemented in Scala, a high-level, functional and object-oriented programming language running on the Java Virtual Machine (JVM). Lisa's interface comes with a domain-specific language (DSL) that enables natural, readable proof scripts in Lisa. Proofs in Lisa use a forward-style syntax (e.g., "have statement by tactic"), in a style similar to Isar proofs in Isabelle. Yet, proof scripts in Lisa are not interpreted like in most other proof assistants: they are fully executable Scala code. A user-written proof is any piece of code that will compute a kernel proof. Tactics are then simply functions computing a proof. Moreover, the proof

DSL (*have* commands) can itself be used in tactics, and can freely be mixed with regular programming constructs like loops, recursion, control flow operators and with the entire standard libraries of Scala and Java. This makes writing new complex tactics in Lisa particularly easy, without the need for plugins or extensions in a separate programming language. Lean [27] is the only other proof assistant we are aware of that has a similar design, with a single unified implementation and tactic language that also offers a human-readable DSL. The crucial difference is that in Lean, that language is Lean itself. This is an impressive feat that allows the user to reason about tactics themselves, but which is incompatible with a small, axiomatic logical foundation such as first-order ZFC.

This chapter is organized as follows: Section 3.1 describes λFOL, a syntactic extension of first-order logic that makes Lisa's logical foundations. Section 3.2 illustrates applications of λFOL to set theory which have been implemented in Lisa's library. Section 3.3 presents the design of Lisa's logical kernel, which guarantees the soundness of the system. Section 3.4 describes Lisa's high-level interface, including its proof script language and tactic framework, and the embedding of λFOL in Scala's type system. Section 3.5 shows how higher-order logic (and in particular proofs from HOL Light) can be efficiently embedded in Lisa's first-order set-theoretic foundations. Finally, Section 3.6 presents SC-TPTP, a proof exchange format designed to facilitate interoperability between proof systems, and in particular between ATPs and ITPs, including Lisa.

3.1 λFOL: First-order logic with functions

First-order logic is the canonical logic used to express mathematical statements and reasoning. It has a simple syntax and deduction system, well-studied proof theory, and, by assuming adequate axioms, is suitable to formalize essentially all concepts of mathematics. However, first-order logic is typically not used in practice to build interactive theorem provers or to reason about programs, as its syntax lacks the expressiveness needed to represent functions and types in a natural way.

In fact, various expressions often used in mathematics simply cannot be represented in first-order logic. One such example is the integral of a function, such as:

$$\int_a^b x^2 \, dx$$

Let's try to formally represent this expression in pure FOL. First note that the result of the integral is a real number, so the expression must be a term (and not a formula). The integral can be thought of as an operator taking three arguments: the lower bound a, the upper bound b, and the function $x \mapsto x^2$. However, functions in first-order logic cannot take other functions as arguments. More generally, the term $\int_a^b x^2 \, dx$ contains the variable x as a subterm, but it is not a *free* variable of the term. As terms can never bind free variables in pure FOL, it is impossible to represent such an expression. In higher-order logic, we could instead define the operator $\int : \text{Ind} \to \text{Ind} \to (\text{Ind} \to \text{Ind}) \to \text{Ind}$

and formally write:

$$\int a\ b\ (\lambda x.x^2)$$

Similarly, consider the standard denotation of set comprehension such as in:

$$\{x \in \mathbb{R} \mid x^2 < 2\}$$

Here again, the syntactically correct equivalent in HOL would be, for a fixed function $\mathcal{C} : \text{Ind} \to (\text{Ind} \to \text{Prop}) \to \text{Ind}$ corresponding to comprehension:

$$\mathcal{C}\ \mathbb{R}\ (\lambda x.x^2 < 2)$$

which again cannot be expressed in pure FOL.

Of course, in a sufficiently powerful theory such as set theory, functions and predicates are also terms themselves, as long as their domain is itself a set. In that case, we can type $\mathcal{C} : \text{Ind} \to \text{Ind} \to \text{Ind}$, and obtain the property that

$$x \in (\mathcal{C}\ A\ P) \iff x \in A \wedge P(x) = \text{true}$$

where A is the base set and P is a set-theoretic function from A to the two-element set $\{\text{true}, \text{false}\}$. In fact, we can even represent predicates by a set directly: then, $\mathcal{C}\ A\ P = A \cap P$. Here, however, we have simply moved the problem around: how do we obtain a term denoting the set-like predicate P from an actual formula such as $x^2 < 2$? Formally, we need an expression that takes a base set A and a predicate P and returns the subset of A satisfying P, which is exactly set comprehension: we went in circles. We can fundamentally not avoid the constraint that a term in first-order logic *cannot* bind free variables nor contain subexpressions which are formulas.

By the comprehension axiom of set theory (or the definition of the operator $\int$), the elements that each of these two expressions denote are well-defined, and they can be shown to exist in pure FOL with ZF axioms, and FOL is complete, so any theorem that mentions such an impossible term has an equivalent. However, compensating for this lack of syntactic expressiveness requires complex encodings. This is not an obstacle to vernacular mathematics, which seldom worries about formal correctness, but is a major obstacle to practical interactive theorem proving, as such encodings are inefficient, difficult to implement and maintain for developers and difficult to reason with for users.

This is the principal reason that has led to the dominance of higher-order logic and type theory-based interactive theorem provers. Systems still relying on first-order logic, such as Mizar [69], Isabelle/FOL [75] and TLAPS [22], all develop their own additional syntax, relying on either a higher-order meta-language, language extensions such as let bindings or implementing set-theoretic axioms directly in higher-order logic (see also [13, 12]).

We present a new formal theory called λFOL. λFOL is a purely logical extension of first-order logic, which allows us to study its proof theory and relations with FOL,

HOL and their variants. The achievement of λFOL is to provide a natural syntax for higher-order expressions, such as in the examples above, while fundamentally remaining first-order and not increasing the complexity of the logical kernel. In fact, it even reduces the complexity because terms and formulas are now treated uniformly.

Definition 3.1.1 (λFOL)**.** *Types*, denoted by σ, τ, are *Terms*, *Formulas* and recursively built function types:

$$\tau := \text{Ind} \mid \text{Prop} \mid \tau_1 \to \tau_2$$

The set of typed expressions is the same as that of simply typed λ-calculus, with four constructors: variable symbols, constant symbols, abstraction and application.

$$\begin{aligned}
&(\mathcal{E} : \tau) := \\
&\mid x_\tau : \tau \\
&\mid c_\tau : \tau \\
&\mid (f : \tau_1 \to \tau_2)(e : \tau_1) : \tau_2 \\
&\mid \lambda x : \tau_1.\ (e : \tau_2) : \tau_1 \to \tau_2
\end{aligned}$$

Variables x_τ range over a countably infinite set and c's are constant symbols. Note that the type of variables is always explicitly attached to the variable symbol, so that any expression always has a single type, independently of any context.

Constant symbols of λFOL are the usual connectives of first-order logic:

$$\begin{aligned}
\neg : &\quad \text{Prop} \to \text{Prop} \\
\wedge : &\quad \text{Prop} \to \text{Prop} \to \text{Prop} \\
\vee : &\quad \text{Prop} \to \text{Prop} \to \text{Prop} \\
\forall, \exists : &\quad (\text{Ind} \to \text{Prop}) \to \text{Prop} \\
\equiv : &\quad \text{Ind} \to \text{Ind} \to \text{Prop} \\
f_1, f_2, ... : &\quad \text{Ind} \to ... \to \text{Ind} \to \text{Ind} \\
P_1, P_2, ... : &\quad \text{Ind} \to ... \to \text{Ind} \to \text{Prop}
\end{aligned}$$

where $\equiv$ is the symbol for equality, $f_1, f_2, ...$ stand for the function symbols of the underlying first-order theory and $P_1, P_2, ...$ for the predicate symbols. For example in Peano arithmetic, we would have three function symbols:

$$\begin{aligned}
&0 : \text{Ind} \\
&S : \text{Ind} \to \text{Ind} \\
&+ : \text{Ind} \to \text{Ind} \to \text{Ind}
\end{aligned}$$

There is a natural embedding of pure FOL terms and formulas in λFOL given by

the injection $|\cdot|_\lambda$:

$$
\begin{aligned}
|x|_\lambda &:= x : \text{Ind} \\
|c|_\lambda &:= c : \text{Ind} \\
|f(t_1, \ldots, t_n)|_\lambda &:= f(|t_1|_\lambda) \ ... \ (|t_n|_\lambda) \\
|P(t_1, \ldots, t_n)|_\lambda &:= P(|t_1|_\lambda) \ ... \ (|t_n|_\lambda) \\
|\neg\phi|_\lambda &:= \neg|\phi|_\lambda \\
|\phi_1 \wedge \phi_2|_\lambda &:= \wedge \ |\phi_1|_\lambda \ |\phi_2|_\lambda \\
|\phi_1 \vee \phi_2|_\lambda &:= \vee \ |\phi_1|_\lambda \ |\phi_2|_\lambda \\
|\forall x.\phi|_\lambda &:= \forall(\lambda x : \text{Ind}.\ |\phi|_\lambda) \\
|\exists x.\phi|_\lambda &:= \exists(\lambda x : \text{Ind}.\ |\phi|_\lambda) \\
|\phi_1 \equiv \phi_2|_\lambda &:= \equiv \ |\phi_1|_\lambda \ |\phi_2|_\lambda
\end{aligned}
$$

We say of λ-expressions in the range of $|\cdot|_\lambda$ that they are *pure first-order expressions*. We often use the syntax of pure FOL for binders and infix conjunction and disjunction to describe the corresponding λFOL expressions.

Definition 3.1.2 (Terms, formulas and order)**.** Expressions of type Prop are called *formulas*. Expressions of type Ind are called *terms*. Expressions of type Ind → Ind → ... → Ind are called *functionals*. Expressions of type Ind → Ind → ... → Prop are called *predicates*. Formulas and terms are said to be of *first-order*. Predicate and function symbols are said to be of *second order*. Generally, define the order of a type as:

$$
\begin{aligned}
\text{ord}(\text{Ind}) &:= 1 \\
\text{ord}(\text{Prop}) &:= 1 \\
\text{ord}(\tau_1 \to \tau_2) &:= \max(\text{ord}(\tau_1) + 1, \text{ord}(\tau_2))
\end{aligned}
$$

If $e : \tau$ is an expression, $\text{ord}(e) := \text{ord}(\tau)$. For example, $\text{ord}(\forall) = \text{ord}(\exists) = 3$ and $\text{ord}(\neg) = \text{ord}(\wedge) = \text{ord}(\vee) = \text{ord}(\equiv) = 2$. For f a function symbol, $\text{ord}(f) = 1$ if f has arity 0 (i.e. it is a true constant) and $\text{ord}(f) = 2$ otherwise. Hence, in Peano arithmetic, $\text{ord}(0) = 1$, $\text{ord}(S) = 2$ and $\text{ord}(+) = 2$.

Definition 3.1.3 (Free variables in λFOL)**.** A variable x is free in a λFOL expression if and only if it follows from the following rules:

$$
\begin{aligned}
x \text{ is free in } x \\
x \text{ is free in } e_1 \ e_2 \text{ if } x \text{ is free in } e_1 \text{ or in } e_2 \\
x \text{ is free in } \lambda y : \text{Ind}.\ e \text{ if } x \neq y \text{ and } x \text{ is free in } e
\end{aligned}
$$

A variable is bound in an expression if it is a subtree of this expression and not free. A variable is *fresh* for a formula if it is not a subterm of the formula (neither free nor

bound).

Definition 3.1.4 ($\alpha\beta\eta$-equivalence)**.** $\sim_\alpha$, $\sim_\beta$ and $\sim_\eta$ are the least congruence relations on expressions containing respectively the following rules:

$$\begin{aligned} \lambda x.\ e &\sim_\alpha \lambda y.\ e[x := y] && \text{(if } y \text{ is not free in } e\text{)} \\ (\lambda x.\ e)\ t &\sim_\beta e[x := t] \\ \lambda x.\ (e\ x) &\sim_\eta e && \text{(if } x \text{ is not free in } e\text{)} \end{aligned}$$

$\alpha\beta\eta$-equivalence, denoted $\sim_{\alpha\beta\eta}$, is the smallest congruence relation containing $\sim_\alpha$, $\sim_\beta$ and $\sim_\eta$.

Definition 3.1.5 (Capture-avoiding substitution in λFOL)**.** The capture-avoiding substitution of a variable x by a term s inside a term t or formula ϕ is denoted $t[x := s]$ or $\phi[x := s]$ and defined recursively as:

$$\begin{aligned} (x)[x := s] &:= s && \\ (y)[x := s] &:= y && \text{if } x \neq y \\ (f(e_1, \ldots, e_n))[x := s] &:= f(e_1[x := s], \ldots, e_n[x := s]) && \\ (P(e_1, \ldots, e_n))[x := s] &:= P(e_1[x := s], \ldots, e_n[x := s]) && \\ (\neg\phi)[x := s] &:= \neg\phi[x := s] && \\ (\phi_1 \wedge \phi_2)[x := s] &:= \phi_1[x := s] \wedge \phi_2[x := s] && \\ (\phi_1 \vee \phi_2)[x := s] &:= \phi_1[x := s] \vee \phi_2[x := s] && \\ (Qx.e_1)[x := e] &:= Qx.e_1 && \\ (Qy.e_1)[x := e] &:= Qy.e_1[x := e] && \text{if } x \neq y \text{ and } y \text{ is not free in } e \\ (Qy.e_1)[x := e] &:= Qz.e_1[y := z][x := e] && \text{otherwise, with } z \text{ fresh} \end{aligned}$$

Where Q stands for either $\forall$ or $\exists$.

Two terms or formulas are *alpha-equivalent* if they differ only by the names of their bound variables. For example, $\forall x.\ P(x)$ and $\forall y.\ P(y)$ are alpha-equivalent. Formally:

$$\begin{aligned} P(t_1, \ldots, t_n) &\equiv_\alpha P(t'_1, \ldots, t'_n) && \text{if } t_i = t'_i \text{ for all } i \\ \neg\phi &\equiv_\alpha \neg\phi' && \text{if } \phi \equiv_\alpha \phi' \\ \phi_1 \wedge \phi_2 &\equiv_\alpha \phi'_1 \wedge \phi'_2 && \text{if } \phi_i \equiv_\alpha \phi'_i \text{ for } i = 1, 2 \\ \phi_1 \vee \phi_2 &\equiv_\alpha \phi'_1 \vee \phi'_2 && \text{if } \phi_i \equiv_\alpha \phi'_i \text{ for } i = 1, 2 \\ \phi_1 \to \phi_2 &\equiv_\alpha \phi'_1 \to \phi'_2 && \text{if } \phi_i \equiv_\alpha \phi'_i \text{ for } i = 1, 2 \\ Qx.e &\equiv_\alpha Qy.e' && \text{if } e[x := y] \equiv_\alpha e' \end{aligned}$$

Unless otherwise specified, we always consider terms and formulas up to alpha-

equivalence. Note that capture-avoiding substitution is compatible with alpha-equivalence in the sense that if $e \equiv_\alpha e'$ then $e[x := s] \equiv_\alpha e'[x := s]$.

Definition 3.1.6 (Mathematical theory)**.** A mathematical theory $\mathbb{T}$ is a pair (Σ, A) where Σ is a FOL signature and A is a set of FOL formulas over Σ, called the *axioms* of $\mathbb{T}$.

Definition 3.1.7 (Sequent calculus for λFOL)**.** First-order logic admits various equivalent deductive systems. For λFOL, we adopt sequent calculus, as it is the foundation of Lisa. Formally, a *sequent* is a pair (Γ, Δ) where Γ and Δ are finite sets of formulas, denoted $\Gamma \vdash \Delta$. A proof in SC^1_λ is a finite rooted tree built from the deduction rules shown in Figure 3.1. We say that the proof proves the sequent at its root. A formula ϕ is *provable* or *valid* if there exists a proof of the sequent $\vdash \phi$.

λFOL seems more expressive than pure first-order logic, as we can write formulas such as (with $S : \mathrm{Ind} \to \mathrm{Ind}$):

$$(\lambda h : (\mathrm{Ind} \to \mathrm{Ind}) \to \mathrm{Ind} \to \mathrm{Prop}.\ \forall(h\ S))(\lambda g : \mathrm{Ind} \to \mathrm{Ind}.\ \lambda x : \mathrm{Ind}.\ (x \equiv g\ x)) : \mathrm{Prop}$$

However, computing the canonical form of this expression yields:

$$\forall\ (\lambda x : \mathrm{Ind}.\ (x \equiv S\ x))$$

corresponding directly to the pure FOL formula $\forall x.\ x \equiv S(x)$. This is in fact a general theorem, *if the formula does not contain free variables of order greater than 1.*

Theorem 3.1.8. Let ϕ be a formula of λFOL, that does not contain free variables of order greater than 1. Then the canonical form of ϕ (denoted ϕ') is a formula of pure first-order logic.

Proof. We analyse the position of λ-abstractions in the expression tree of ϕ'. Consider one of the topmost subexpressions e of ϕ' of the form $e = \lambda x.\ e'$. Note that e cannot contain free variables of order greater than 1, as otherwise they would either be free or bound in ϕ', and by assumption we reject both. Note also that if $\phi' = e$, then ϕ' cannot have type Prop, so e is necessarily a proper subexpression of ϕ'. Hence, e necessarily occurs as part of an application.

- Case 1: e is left of the application. In that case ϕ' contains a subterm of the form $(\lambda x.\ e')t$, which is a β-redex, contradicting the assumption that ϕ' is in canonical form.
- Case 2: e is right of the application, so ϕ' contains a subterm of the form $t\ e$. Note that $\mathrm{ord}(t) \geq 3$. If t is a syntactic abstraction, we once again have a β-redex. Hence, it must be that t is either a variable, an application or a constant symbol. It cannot be a variable since ϕ' does not contain higher-order variables. t cannot be an application either: indeed, let f and $t_1, \ldots, t_n$ be such that $t = f\ t_1\ \ldots\ t_n$.

$$\frac{}{\Gamma, \phi \vdash \phi, \Delta}\ \textsc{Hyp} \qquad \frac{}{\vdash \phi}\ \textsc{Axiom}^{\cdot}\ (\text{if } \phi \in A)$$

$$\frac{\Gamma \vdash \phi, \Delta \qquad \Sigma, \phi \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}\ \textsc{Cut}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma' \vdash \Delta'}\ \textsc{Weakening}^{\cdot}\ (\text{if } \Gamma \subseteq \Gamma' \text{ and } \Delta \subseteq \Delta')$$

$$\frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}\ \textsc{LeftAnd} \qquad \frac{\Gamma \vdash \phi, \Delta \qquad \Sigma \vdash \psi, \Pi}{\Gamma, \Sigma \vdash \phi \wedge \psi, \Delta, \Pi}\ \textsc{RightAnd}$$

$$\frac{\Gamma, \phi \vdash \Delta \qquad \Sigma, \psi \vdash \Pi}{\Gamma, \Sigma, \phi \vee \psi \vdash \Delta, \Pi}\ \textsc{LeftOr} \qquad \frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta}\ \textsc{RightOr}$$

$$\frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta}\ \textsc{LeftNot} \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\ \textsc{RightNot}$$

$$\frac{\Gamma, \phi_x[x := t] \vdash \Delta}{\Gamma, \forall x.\ \phi_x \vdash \Delta}\ \textsc{LeftForall} \qquad \frac{\Gamma \vdash \phi_x, \Delta}{\Gamma \vdash \forall x.\ \phi_x, \Delta}\ \textsc{RightForall}$$

$$\frac{\Gamma, \phi_x \vdash \Delta}{\Gamma, \exists x.\ \phi_x \vdash \Delta}\ \textsc{LeftExists} \qquad \frac{\Gamma \vdash \phi_x[x := t], \Delta}{\Gamma \vdash \exists x.\ \phi_x, \Delta}\ \textsc{RightExists}$$

$$\frac{\Gamma, \phi[x := t] \vdash \Delta \qquad \Sigma \vdash t \equiv u, \Pi}{\Gamma, \Sigma, \phi[x := u] \vdash \Delta, \Pi}\ \textsc{LeftSubst}$$

$$\frac{\Gamma \vdash \phi[x := t], \Delta \qquad \Sigma \vdash t \equiv u, \Pi}{\Gamma, \Sigma \vdash \phi[x := u], \Delta, \Pi}\ \textsc{RightSubst}$$

$$\frac{}{\vdash t \equiv t}\ \textsc{Refl}$$

Figure 3.1: The deduction rules of FOL sequent calculus. Note that in RightForall and RightExists, the variable x must not be free in the resulting sequent.

Then f cannot be an abstraction (as it would imply that ϕ' contains a β-redex), it cannot be a variable since ϕ' does not contain higher-order variables, and it cannot be a constant symbol because in our language there is no constant that matches the type of f. Hence, t must be a higher-order constant symbol, and the only such symbols are $\forall$ and $\exists$. Hence, we necessarily have

$$t\ e = \forall\ (\lambda x : \text{Ind}.\ (e' : \text{Prop})) \sim \forall x.\ e'$$

or

$$t\ e = \exists\ (\lambda x : \text{Ind}.\ (e' : \text{Prop})) \sim \exists x.\ e'$$

Since e' does not contain free variables of order greater than 1, we conclude by induction that e' is a formula of pure first-order logic, and hence ϕ' is a formula of pure first-order logic.

□

Theorem 3.1.9. Let s be a sequent that does not contain free variables of order greater than 1. Then s is provable in SC^1_λ if and only if its canonical form is provable in pure FOL.

Proof. Let s' be the canonical form of s, which by Theorem 3.1.8 is a pure FOL sequent. Replacing a formula by an $\alpha\beta\eta$-equivalent one preserves the applicability of the rules in Figure 3.1. Hence, if s' is provable in SC^1_λ then every intermediate step can be replaced by its canonical form and so s' has a proof where all formulas are pure FOL formulas. Such a proof is also a proof in pure FOL. Conversely, if s' is provable in pure FOL, then it is provable in SC^1_λ since SC^1_λ extends pure FOL, and hence s is provable in SC^1_λ. □

Theorem 3.1.8 tells us that all the new formulas of λFOL are $\alpha\beta\eta$-equivalent to some pure first-order formula. It hence seems that we did nothing but create new representations for pure first-order formulas. That is true, but only because we explicitly disallowed free higher-order variables and constant symbols. Theorem 3.1.9 should offer a first insight about why we claim that λFOL is not more powerful than pure first-order logic. In the subsequent subsections, we will discuss, on the other hand, how to use free higher-order variables and constant symbols to increase expressiveness.

3.1.1 Higher-order variables

First-order logic by itself does not allow one to formalize much mathematics, and needs the axiomatization of some mathematical theory. ZF set theory, with or without the axiom of choice, will be our theory of choice to lay the ground of all mathematics, but we can also consider simpler theories such as Peano arithmetic. Both of these theories have the shared property that they cannot be finitely axiomatized in pure FOL. For example, Peano arithmetic contains the axiom schema of induction:

$$P(0) \wedge (\forall x.\ P(x) \implies P(S(x))) \implies \forall x.\ P(x)$$

This schema is really an infinite set of axioms, one for every different predicate P of FOL with symbols $0, S, +$. It is hence convenient, and common in practice, to augment pure FOL with *schematic functions and predicates*, such as the predicate P in the axiom schema of induction. These symbols have the property that they can be instantiated with any first-order expression, for example:

$$\begin{aligned} & (\forall y.\ y + 0 = 0 + y) \land (\forall x.\ (\forall y.\ y + x = x + y) \implies \\ \Gamma \quad \vdash\ & (\forall y.\ y + S(x) = S(x) + y)) \implies \\ & \forall x.\ (\forall y.\ y + x = x + y) \end{aligned}$$

is the result of instantiating P with the one-parameter formula $x \mapsto \forall y.\ y + x = x + y$. This coincides with the usual instantiation of variables in higher-order logic. Of course, there is no reason to restrict schematic symbols to only appear in axioms. Then, we can use them to prove theorem schemas that cannot be expressed in pure FOL, such as:

$$\forall x, y.\ x = y \implies (P(x) \iff P(y))$$

or properties of logical connectives such as:

$$(A \land (B \lor C)) \iff ((A \land B) \lor (A \land C))$$

Note that free variables can be seen as a special case of schematic function symbols of arity 0. Then, free higher-order variables in λFOL are a generalization of schematic function and predicate symbols beyond order 2! Intuitively, statements containing free higher-order variables represent at once all possible ground instantiations of the statement. The following theorem formalizes this intuition.

Definition 3.1.10. The system SC^2_λ comprises the same deduction rules as SC^1_λ, with the addition of the following rule:

$$\frac{\Gamma \vdash \Delta}{\Gamma[f := e] \vdash \Delta[f := e]}\ \textsc{Inst}$$

Where f is a (possibly higher-order) variable of type τ and e any expression of type τ.

Theorem 3.1.11. Let s be a sequent where all formulas are in pure FOL. Then s is provable in SC^2_λ if and only if it is provable in SC^1_λ (and hence in pure FOL).

Proof. The "if" direction is trivial, as SC^1_λ is a subsystem of SC^2_λ. For the "only if" direction, first rename variables across the proof so that each variable is only instantiated once. Then the proof is by induction on the size of the proof of s in SC^2_λ. We proceed by case analysis on the last step of the proof, and note that the only new rule and non-trivial case is Inst.

$$\frac{\dfrac{\mathcal{A}}{\Gamma \vdash \Delta}}{\Gamma[f := e] \vdash \Delta[f := e]}\ \textsc{Inst}$$

Now, replace every occurrence of f in every formula of every sequent of $\mathcal{A}$ by e, normalizing the result to canonical form. The result is still a valid proof of $\Gamma[f := e] \vdash \Delta[f := e]$ in SC^2_λ, but of size one lower than the original proof. By induction, there is a proof of $\Gamma \vdash \Delta$ in SC^1_λ. □

3.1.2 Constants, definitions and description operators

To make use of higher-order constant symbols, we need a definition principle to introduce new symbols in a mathematical theory. We will adopt the simplest possible such principle: a new constant is introduced as a definitional abbreviation for a closed expression.

Definition 3.1.12. For a theory $\mathbb{T}$, a *definition* is a pair (c, e) where $c : \tau$ is a constant symbol that is not in the language of $\mathbb{T}$ and e is an expression of type τ that does not contain free variables. If $\tau = \sigma_1 \to ... \to \sigma_n \to \mathrm{Ind}$, the theory $\mathbb{T}'$ resulting from the definition (c, e) is the theory $\mathbb{T}$ extended with the constant symbol $c : \tau$ and the axiom $(cAB...) \equiv (eAB...)$, where $A, B, ...$ are variables of adequate types $\sigma_1, \ldots, \sigma_n$. Otherwise if $\tau = \sigma_1 \to ... \to \sigma_n \to \mathrm{Prop}$, the added axiom is instead $(cAB...) \iff (eAB...)$.

This definition principle is rather weak. In contrast, the usually allowed definition principle for pure FOL is as follows: If $\forall x, y, ... \exists x.\ P(x)$ is a proven theorem, then we may introduce a function symbol f and the axiom $\forall x, y, ... P(f(x, y, ...))$. This principle allows giving names to objects that exist but which have no description, and is similar to the process of skolemization in automated theorem proving. It can be shown that this principle yields a conservative extension [83]. The following trivial example illustrates the difference between the two definition principles.

Example 3.1.13. Let $\mathbb{T}$ be a FOL theory with no function symbol, one predicate symbol P of arity 1, and only a single axiom, stating $\exists x.\ P(x)$. Since the language contains no constant symbol, every term must contain a free variable and hence our first definition principle can never be used. In contrast, the standard definition principle allows one to define a symbol c and the axiom $P(c)$.

Moreover, even with a less trivial theory, we still have not addressed one key obstacle in representing expressions such as the integral or set comprehension. By Theorem 3.1.8, every term that does not contain free higher-order variables is equivalent to a pure FOL term, which still cannot bind free variables. The latter definition principle allows to indirectly build terms from formulas (which can bind variables), but we do not want to introduce a new definition for every anonymous function. We can solve this problem as well as the problem of defining new constant symbols using description operators.

Definition 3.1.14 ($\mathrm{SC}^\epsilon_\lambda$ and $\mathrm{SC}^\iota_\lambda$)**.**

(Indefinite Description). The system $\mathrm{SC}^\epsilon_\lambda$ is the same as SC^2_λ, with the addition of a new constant symbol ϵ of type $(\mathrm{Ind} \to \mathrm{Prop}) \to \mathrm{Ind}$ and the following deduction rule:

$$\frac{\Gamma \vdash \exists x.\ \phi_x}{\Gamma \vdash \phi[x := \varepsilon x.\ \phi_x]}\ \textsc{Epsilon}$$

(Definite Description). Similarly, the system $\mathrm{SC}^{\iota}_{\lambda}$ is the same as $\mathrm{SC}^{2}_{\lambda}$, with the addition of a new constant symbol ι of type $(\mathrm{Ind} \to \mathrm{Prop}) \to \mathrm{Ind}$ and the following deduction rule:

$$\frac{\Gamma \vdash \exists! x.\ \phi_x}{\Gamma \vdash \phi[x := \iota x.\ \phi_x]}\ \textsc{Iota}$$

The only difference between the two systems is that in $\mathrm{SC}^{\iota}_{\lambda}$ the premise of the Iota rule requires existence and uniqueness, while in $\mathrm{SC}^{\epsilon}_{\lambda}$, Epsilon requires existence only.

In pure FOL, ϵ and ι are binders for free variables. The expression $\varepsilon x.\ P(x)$ is a term that denotes some element that satisfies P, if one exists. If no such element exists, then nothing can be deduced about $\varepsilon x.\ P(x)$ and it can denote any term in the domain. Similarly, $\iota x.\ P(x)$ denotes the unique element that satisfies P, if it exists, and otherwise it can denote any term in the domain. It has been shown that extending first-order logic with either ϵ or ι is conservative, a fact known as the epsilon theorems [92]. However, in the presence of axiom schemas, only ι is conservative.

Lemma 3.1.15. In ZF set theory, if formulas containing ϵ are allowed to instantiate axiom schemas, the axiom of choice is provable.

Proof. Consider the term $\varepsilon x.\ x \in y$, parametrized by y. For any set $y \neq \emptyset$, this term denotes an element of y. Using this term within the axiom schema of replacement, we can obtain for any set A containing only non-empty sets, the set:

$$\{(x, \varepsilon y.\ y \in x) \mid x \in A\}$$

which is a choice function for A. Hence, using the new symbol, we can prove the axiom of choice, which is well known to be independent of ZF, so the extension is not conservative. □

This situation is not specific to the use of epsilon and can be similarly constructed using a principle of definition by existence. Concretely, the formula

$$\forall x.\ \exists y.\ x = \emptyset \lor y \in x$$

is provable in ZF. We can hence define a new constant function $\mathrm{choice}(x)$ with the property $x = \emptyset \lor \mathrm{choice}(x) \in x$. Using this function within the axiom schema of replacement we can similarly obtain a choice function.

This seems paradoxical, since we just claimed that this definition principle is conservative. The resolution of the paradox is that by introducing a new symbol choice, we implicitly also introduce new axioms. There would be no paradox if we treated axioms as a fixed set and the new symbols could not be used to instantiate the axiom schemas.

This is, however, rather unintuitive and impractical for a user of the theory. We hence can either accept having built-in global choice via ϵ, or instead use the weaker ι operator, which yields a stronger property than mere conservativity, which we call *full conservativity*.

Definition 3.1.16. A theory $\mathcal{T}_2$ is a fully conservative extension over a theory $\mathcal{T}_1$ if:

- it is conservative, and
- for any formula ϕ_2 with free variables $x_1, \ldots, x_k$ in the language of $\mathcal{T}_2$, there exists a formula ϕ_1 in the language of $\mathcal{T}_1$ with free variables among $x_1, \ldots, x_k$ such that

$$\mathcal{T}_2 \vdash \forall x_1 ... x_k . (\phi_1 \leftrightarrow \phi_2)$$

Theorem 3.1.17. $\mathrm{SC}^{\iota}_{\lambda}$ is a fully conservative extension over $\mathrm{SC}^{2}_{\lambda}$.

Proof. Since $\mathrm{SC}^{\iota}_{\lambda}$ is weaker than $\mathrm{SC}^{\epsilon}_{\lambda}$, by the second epsilon theorem [34], it is a conservative extension of $\mathrm{SC}^{2}_{\lambda}$.

Let $\phi[x := \iota y.\ \psi]$ be a formula that contains a ι expression. We will construct another formula that is equiprovable in $\mathrm{SC}^{2}_{\lambda}$, and that is provably equivalent in $\mathrm{SC}^{\iota}_{\lambda}$. First note that it is clearly equivalent to the formula $\forall z.\ z = \iota y.\ \psi \implies \phi[x := z]$. Then, by the Iota rule, $z = \iota y.\ \psi \iff \psi[y := z]$. Hence, the original formula is equivalent in $\mathrm{SC}^{\iota}_{\lambda}$ to the formula $\forall z.\ \psi[y := z] \implies \phi[x := z]$. This formula contains one fewer occurrence of a ι expression, so we conclude by induction. □

Which of ι and ϵ to use matters little for the application of λFOL. For all our future applications, ι will be sufficient. In Lisa, we adopt ϵ for the very pragmatic reason that it requires less effort to use. ι can then be defined and Iota derived.

Note. By allowing us to use higher-order expressions, λFOL gives us a lot of expressive power. But it raises a question: Are we really not, in fact, doing higher-order logic in disguise, and if not, what differentiates us from higher-order logic? The answer is that we miss both quantification over variables of order greater than 1 and equality on terms of order greater than 2. We do have equality on terms of order 1, mainly $\equiv$ for terms and $\iff$ for formulas. We can also define equality on terms of order 2 by extensionality: for $f, g : \sigma_1 \to ... \to \sigma_n \to \mathrm{Ind}$, we could define

$$f \equiv_2 g := \forall x_1, \ldots, x_n.\ (f\ x_1\ ...\ x_n) \equiv (g\ x_1\ ...\ x_n)$$

Assuming quantification over variables of order n, we can define equality between functions of order $n+1$ similarly. Conversely, given $\equiv_n$ the equality on terms of order n, we can define quantification $\forall_n : \sigma \to \mathrm{Prop}$ where $P : \sigma \to \mathrm{Prop}$ has order n:

$$\forall_n P := P \equiv_n \lambda x : \sigma.\ \top$$

which is the standard way of defining quantification in higher-order logic, for example in HOL Light.

Note, however, that we do have a limited form of higher-order equality: we can express that Prop is extensionally equal to G with the sequent:

$$\vdash (\text{Prop } x_1 ... \ x_n) \equiv_1 (G \ x_1 ... \ x_n)$$

where the x_i's are of order greater than 1 and are free. At the meta-level, free variables are universally quantified, so this statement is in some sense equivalent to higher-order extensional equality. This is the notion of equality we use to define higher-order symbols. However, it is not a true equality: it cannot be embedded inside the language of λFOL, and it is in particular not possible to state such equality as an assumption. Indeed, to see why the explicit quantification is significant, consider the following example: if $\text{Prop}, G : \sigma \to \text{Ind}$, $e : \sigma$ is an arbitrary expression and $x : \sigma$ is a variable not free in e, then the following formula is certainly not a theorem:

$$\text{Prop } x \equiv G \ x \implies \text{Prop } e \equiv G \ e$$

whereas if $x, c : \text{Ind}$, this one certainly is:

$$(\forall x : \text{Ind}. \ \text{Prop } x \equiv G \ x) \implies \text{Prop } e \equiv G \ e$$

The second, important in practice but less crucial difference between HOL and λFOL is that λFOL does not admit type variables. We do not need them in practice, since all the actual reasoning and processing takes place in the underlying set theory at the term level. The higher-order elements of λFOL only serve a syntactic purpose. Hence, we argue that λFOL is still, in spirit, first-order logic; only equipped with the means to reason about axioms and theorem schemas, and with syntactic tools that match the common vernacular practice of mathematics, and which additionally will be very convenient for mechanized theorem proving, as we will see in the further sections.

3.2 Applications to set theory

Equipped with λFOL, we can now properly define higher-order symbols and binders, such as set comprehension. Concretely, we consider λFOL alongside the ten axioms of ZFC as defined in Lisa (Figure 3.2).

Definition 3.2.1 (λFOST)**.** λFOST is the extension of λFOL whose constant symbols include all those of λFOL plus the following:

$$\begin{aligned} \in &: && \text{Ind} \to \text{Ind} \to \text{Prop} \\ \emptyset &: && \text{Ind} \\ \{\cdot,\cdot\} &: && \text{Ind} \to \text{Ind} \to \text{Ind} \\ \bigcup &: && \text{Ind} \to \text{Ind} \\ \mathcal{P} &: && \text{Ind} \to \text{Ind} \\ \subseteq &: && \text{Ind} \to \text{Ind} \to \text{Prop} \end{aligned}$$

and whose axioms are the ten axioms of ZFC as stated in Figure 3.2.

Definition 3.2.2 (Comprehension operator)**.** Define $\mathcal{C} : T \to (T \to F) \to T$ as:

$$\mathcal{C}\ A\ P := \iota x.\ (\forall y.\ y \in x \iff (y \in A \land (P\ y)))$$

Moreover, we write $\mathcal{C}\ A\ P$ using the notation

$$\{x \in A \mid (P\ x)\}$$

Using the comprehension schema, the following (schematic) statement is then a theorem:

$$y \in \{x \in A \mid P\} \iff (y \in A \land (P\ y))$$

$\mathcal{C}$ can be seen as a *filter* operation on the set A by the predicate P. It is then natural to define other common operations on collections, such as *map*:

Definition 3.2.3. Define $\text{map} : T \to (T \to T) \to T$ as:

$$\text{map}\ \ A\ F := \iota x.\ (\forall y.\ (y \in x) \iff (\exists y'.\ y' \in A \land y = (F\ y')))$$

We write $\text{map}\ \ A\ F$ using the notation:

$$\{Fx \mid x \in A\}$$

This time using the replacement schema, we can prove that the desired set always exists and hence obtain the following theorem schema:

$$\forall y.\ y \in \text{map}\ \ A\ F \iff \exists y'.\ y' \in A \land y = (F\ y')$$

extensionalityAxiom	$\vdash \forall z.\ (z \in x \iff z \in y) \iff (x = y)$
pairAxiom	$\vdash z \in \{x, y\} \iff (z = x \lor z = y)$
comprehensionSchema	$\vdash \exists z.\ \forall x.\ x \in z \iff (x \in y \land \varphi(x))$
emptySetAxiom	$\vdash x \notin \emptyset$
unionAxiom	$\vdash z \in \bigcup x \iff \exists y.\ (y \in x \land z \in y)$
subsetAxiom	$\vdash (x \subseteq y) \iff \forall z.\ (z \in x \implies z \in y)$
powerSetAxiom	$\vdash x \in \mathcal{P}(y) \iff x \subseteq y$
infinityAxiom	$\vdash \exists x.\ \emptyset \in x \land \forall y.\ y \in x \implies \bigcup\{y, \{y, y\}\} \in x$
foundationAxiom	$\vdash x \neq \emptyset \implies \exists y \in x.\ \forall z.\ z \in x \implies z \notin y$
replacementSchema	$\vdash \Big(\forall x \in A.\ \forall y, z.\ P(x)(y) \land P(x)(z) \implies y = z\Big) \implies$ $\exists B.\ \forall y.\ y \in B \iff \exists x \in A.\ P(x)(y)$

Figure 3.2: ZFC axioms in Lisa.

From this, we can define similarly flatMap, collect and other similar operations.

$\mathcal{C}$ can be seen as the filter operation on a set, but alternatively, it can also be seen as the restriction of the class P to the set A, in a sense casting a class into a set. Similarly, we would like to cast any functional $T \to T$, which is defined on the entire universe of sets, into a function-like set, which necessarily has a restricted domain.

Definition 3.2.4 (Set-theoretic function)**.** A *set-theoretic function* (or just *function*) is a set f such that:

$$\begin{aligned}&\text{isFunction } f :=\\ &\quad \forall x \in f.\ \exists x_1, x_2.\ x = (x_1, x_2) \land \forall y_1, y_2.\ (y_1, y_2) \in f \implies y_1 = x_1 \implies y_2 = x_2\end{aligned}$$

Define an operator $\text{fun} : T \to (T \to T) \to T$ as:

$$\text{fun}\ \ A\ (F) := \{(x, Fx) \mid x \in A\}$$

We use the alternative notation for $\text{fun}\ \ A\ (\lambda x : T.\ t_x)$:

$$\lambda x \in A.\ t_x$$

Note that this λ-abstraction is different from the λ-abstraction of λFOL: we use $\in$ instead of : to distinguish the two, but the resemblance is very much intentional, as we ultimately want to embed a full type system in set theory.

Define also the application $\text{apply}\ \ : T \to T \to T$ as:

$$\text{apply}\ \ f\ x := \iota y.\ (x, y) \in f$$

We will often write $f@x$ as notation for apply $f\ x$.

We can now use fun to write set-theoretic functions directly as actual terms, without using a definition principle. For example:

$$\lambda x \in \mathbb{R}.\ x^2 - 3 * x + 5$$

Note that this is really a term of type T, that is a set, and not a functional of type $T \to T$. We can also prove the generic theorem schema:

$$y \in A \implies (\lambda x \in A.\ Fx)@y = (F\ y)$$

which precisely corresponds to β-reduction *but this time at the level of set-theoretic functions.*

We then also define $\Rightarrow : T \to T \to T$ (written infix) as:

$$A \Rightarrow B :=$$
$$\{f \in \mathcal{P}(A \times B) \mid \text{isFunction}\, f \wedge (\text{isTotal}\ \ A\ f) \wedge (\text{dom}\, f = A) \wedge (\text{range}\ \ f \subseteq B)\}$$

where $\text{dom} : T \to T$ denotes the domain of f: $\text{dom}\, f := \{(\pi_1\ x) \mid x \in f\}$ and $\text{range} : T \to T$ denotes the range of f: $\text{range}\, f := \{(\pi_2\ x) \mid x \in f\}$.

Theorem 3.2.5. The following statements are theorems:

$$\vdash \text{isFunction}\ (\lambda x \in A.\ F\ x) \tag{3.1}$$
$$\vdash \text{isTotal}\ \ A\ (\lambda x \in A.\ F\ x) \tag{3.2}$$
$$\vdash \text{dom}\ (\lambda x \in A.\ F\ x) = A \tag{3.3}$$
$$f \in A \Rightarrow B \vdash (\text{isFunction}\ \ f \wedge \text{isTotal}\ \ A\ f) \tag{3.4}$$
$$\forall x \in A.\ F\ x \in B \vdash \lambda x \in A.\ F\ x \in A \Rightarrow B \tag{3.5}$$
$$f \in A \Rightarrow B, x \in A \vdash f@x \in B \tag{3.6}$$
$$y \in A \vdash (\lambda x \in A.\ Fx)@y = (F\ y) \tag{3.7}$$
$$f \in A \Rightarrow B, g \in A \Rightarrow B, \forall x \in A. f@x = g@x \vdash f = g \tag{3.8}$$

With this definition of functions, we could define the integral of an arbitrary function. In fact, we do not need λFOL much, beyond comprehension, replacement and fun, as we can do most else directly within T. For the integral:

$$\int : T := \quad \iota f.\ f \in \mathbb{R} \Rightarrow \mathbb{R} \Rightarrow (\mathbb{R} \Rightarrow \mathbb{R}) \Rightarrow \mathbb{R} \wedge (\forall a, b, g.\ \text{isIntegral}\ \ a\ b\ g\ (f@a@b@g))$$

where $\text{isIntegral}\, a\ b\ g\ v$ states that v is the value of the integral of g between a and b (formalizing real analysis is left as an exercise to the reader). What is important to note is that $\int$ is a set-theoretic function and not a functional, and fun alone as a higher-order symbol enables our higher-order syntax. We can hence write $\int_a^b x^2 - 2x + 2\ dx$ as a

notation for $\int @a@b@(\lambda x \in \mathbb{R}.\ x^2 - 2 * x + 2)$.

This is more generally true: most of modern mathematics, beyond foundational set theory itself, can be done using only functions whose domains are sets rather than classes. It is not surprising, as such functions correspond to what can be expressed in higher-order logic, and set theory provides a natural model of higher-order logic where HOL types are sets and HOL functions are set-theoretic functions. We will make use of this correspondence in Subsection 3.5.2 to define a type system for set theory that subsumes HOL. True higher-order functionals are still useful to manipulate classes rather than sets, but again in practice they are more tools rather than objects of study. The next example illustrates various functions and functionals, along with their types (we use the term "type" both to describe the *unique* type of an expression in λFOL, and a *set* of the form $A \Rightarrow B$ containing a set-theoretic function. Note that the latter one is in general not unique, but we purposefully do not discuss subtyping here). Of course, all set-theoretic functions also have type T. Conversely, all functionals of order 2 can be projected to a set-theoretic function by restricting their domain to some set (which can be seen as a form of polymorphism), and the type of formulas F to the set $\{0, 1\}$.

Example 3.2.6.

Set-Theoretic Function	**Set to which it belongs**
$\lambda x \in A.\ x$	$A \Rightarrow A$
$\lambda x \in \mathbb{R}.\ x^2 + x$	$\mathbb{R} \Rightarrow \mathbb{R}$
$+_{\mathbb{N}}$	$\mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$
log	$\mathbb{R}^+ \Rightarrow \mathbb{R}$
$\int$	$\mathbb{R} \Rightarrow \mathbb{R} \Rightarrow (\mathbb{R} \Rightarrow \mathbb{R}) \Rightarrow \mathbb{R}$
$\lambda x \in \mathcal{P}(A).\ A \setminus x$	$\mathcal{P}(A) \Rightarrow \mathcal{P}(A)$
$\lambda x \in G.\ x^{-1}$	$G \Rightarrow G$
$\lambda x \in A \times B.\ \pi_1(x)$	$(A \times B) \Rightarrow A$

Functional	**Type**	**Order**
$\emptyset$	T	1
$\cup, \cap$	$T \to T \to T$	2
$\mathcal{P}$	$T \to T$	2
$\lambda x : T.\ x$	$T \to T$	2
fun	$T \to (T \to T) \to T$	3
apply	$T \to T \to T$	2
$\mathcal{C}$	$T \to (T \to F) \to T$	3
map	$T \to (T \to T) \to T$	3
pair	$T \to T \to T$	2
π_1, π_2	$T \to T$	2
$+_{\text{ord}}$	$T \to T \to T$	2
$\times_{\mathcal{G}}$	$T \to T \to T$	2

x^{-1} denotes inverse in a group, $+_{\mathrm{ord}}$ addition on ordinals, and $\times_{\mathcal{G}}$ the product of two arbitrary groups.

3.3 Designing a proof assistant, part 1: Lisa's kernel

Lisa's kernel is the starting point of Lisa, formalizing the foundations of the whole theorem prover. It is the only trusted code base, meaning that if it is bug-free then no further erroneous code can violate the soundness property and prove invalid statements. Hence, the two main goals of the kernel are to be efficient and trustworthy.

Lisa's foundations are based on the traditional (in the mathematical community) foundational theory of all mathematics, but with some extensions and modifications to more closely match common mathematical practice and improve efficiency and usability:

- The syntax of Lisa's statements is built upon λ**FOL** as described in Section 3.1.
- The deductive system of Lisa's kernel is **Sequent Calculus**.
- The axiomatic theory is **ZFC Set Theory**, but the kernel is actually theory-agnostic and is sound to use with any other set of axioms.

3.3.1 Lisa's syntax

The basic elements of Lisa are called *Expressions*, generalizing terms and formulas. Expressions are terms of the simply typed λ-calculus, with two basic types: *Prop*, or propositions, corresponding to formulas, and *Ind*, or individuals, corresponding to terms. To disambiguate from Scala types and set-theoretic types, we call Prop and Ind *Sorts*. Formally:

Definition 3.3.1 (Identifiers)**.** Identifiers are pairs of strings and non-negative integers used to name symbols

```
case class Identifier(name: String, no: Int)
```

Identifiers cannot contain the symbols `()[]{}?,;_`` nor whitespace. The canonical representation of an identifier is

$$\text{ID}(\texttt{"foo"}, i) = \begin{cases} \texttt{foo} & \text{if } i = 0 \\ \texttt{foo\_}i & \text{else} \end{cases}$$

Including an explicit counter in identifiers allows more efficient generation of fresh symbols, as we can simply increment the counter instead of searching for unused strings.

Sorts and expressions are defined as in Definition 3.1.1. Concretely:

```
case object Ind extends Sort
case object Prop extends Sort
case class Arrow(from: Sort, to: Sort) extends Sort

case class Variable(id: Identifier, sort: Sort)         extends Expression
case class Constant(id: Identifier, sort: Sort)         extends Expression
case class Application(f: Expression, arg: Expression)  extends Expression
case class Lambda(v: Variable, body: Expression)        extends Expression
```

Every expression must belong to a sort. Ill-sorted expressions are forbidden.

	To decide...	Try...
1	$\{\wedge, \vee, \to, \leftrightarrow, \neg\}(\vec{\phi}) \leq \psi$	Base *OL* algorithm
2	$\phi \leq \{\wedge, \vee, \to, \leftrightarrow, \neg\}(\vec{\psi})$	Base *OL* algorithm
3	$s_1 \equiv s_2 \leq t_1 \equiv t_2$	$\{s_1, s_2\} = \{t_1, t_2\}$
4	$\phi \leq t_1 \equiv t_2$	$t_1 = t_2$
5	$\forall.\phi \leq \forall.\psi$	$\phi \leq \psi$
6	$C(\phi_1, \ldots, \phi_n) \leq C(\psi_1, \ldots, \psi_n)$	$\phi_i \sim \psi_i$, for every $1 \leq i \leq n$
7	Anything else	`false`

Table 3.1: Extension of the *OL* algorithm to first-order logic. We call it the F(OL)^2 algorithm. $\equiv$ denotes the equality predicate in FOL, while $=$ denotes syntactic equality of terms.

3.3.2 Lisa's equivalence checker

While proving theorems, trivial syntactical transformations such as $p \wedge q \rightsquigarrow q \wedge p$ increase the length of proofs, which is desirable to neither the user nor the machine. Moreover, the proof checker will very often have to check whether two formulas that appear in different sequents are the same. Hence, instead of using pure syntactical equality, Lisa implements an equivalence checker able to detect a class of equivalence-preserving logical transformations. For example, we would like the formulas $p \wedge q$ and $q \wedge p$ to be naturally treated as equivalent, and similarly for the terms $(\lambda x : T.x)\ y$ and y.

This equivalence checker first beta-normalizes expressions, expresses quantified variables using de Bruijn indices, and desugars $\exists.\phi$ into $\neg\forall.\neg\phi$, $\phi \Leftrightarrow \psi$ into $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$, and $\phi \Rightarrow \psi$ into $\neg\phi \vee \psi$. It then applies *OL* normalization, with the additional rules of Table 3.1.

Equivalence as determined by the equivalence checker is similar in spirit to convertibility in proof assistants such as Rocq and Lean; as far as the kernel is concerned, two equivalent expressions are interchangeable, and a proof will always stay correct if an expression is replaced by an equivalent one.

3.3.3 Proofs in sequent calculus for λFOL

The deductive system used by Lisa is an extended version of the sequent calculus $\mathrm{SC}^{\epsilon}_{\lambda}$, as described in Section 3.1, extended with rules for $\exists!$, $\Longrightarrow$, $\Longleftrightarrow$ and a RESTATE rule leveraging the F(OL)^2 algorithm from Subsection 3.3.2. The detailed list of Lisa's proof steps is given in Figure 3.4.

A sequent calculus proof is a tree whose nodes are proof steps. The root of the proof shows the concluding statement, and the leaves are either assumptions (for example, set-theoretic axioms) or proof steps taking no premise (Hypothesis, RIGHTREFL and RESTATETRUE). Figure 3.5 shows an example of a proof tree for Peirce's Law in strict Sequent Calculus.

In Lisa's kernel, proof steps are organized linearly, in a list, to form actual proofs.

$$\frac{}{\Gamma, \phi \vdash \phi, \Delta}\ \textsc{Hyp} \qquad \frac{\Gamma \vdash \phi, \Delta \qquad \Sigma, \phi \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}\ \textsc{Cut}$$

$$\frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}\ \textsc{LeftAnd} \qquad \frac{\Gamma \vdash \phi, \Delta \qquad \Sigma \vdash \psi, \Pi}{\Gamma, \Sigma \vdash \phi \wedge \psi, \Delta, \Pi}\ \textsc{RightAnd}$$

$$\frac{\Gamma, \phi \vdash \Delta \qquad \Sigma, \psi \vdash \Pi}{\Gamma, \Sigma, \phi \vee \psi \vdash \Delta, \Pi}\ \textsc{LeftOr} \qquad \frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta}\ \textsc{RightOr}$$

$$\frac{\Gamma \vdash \phi, \Delta \qquad \Sigma, \psi \vdash \Pi}{\Gamma, \Sigma, \phi \to \psi \vdash \Delta, \Pi}\ \textsc{LeftImplies} \qquad \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \to \psi, \Delta}\ \textsc{RightImplies}$$

$$\frac{\Gamma, \phi \to \psi \vdash \Delta}{\Gamma, \phi \leftrightarrow \psi \vdash \Delta}\ \textsc{LeftIff} \qquad \frac{\Gamma \vdash \phi \to \psi, \Delta \qquad \Sigma \vdash \psi \to \phi, \Pi}{\Gamma, \Sigma \vdash \phi \leftrightarrow \psi, \Delta, \Pi}\ \textsc{RightIff}$$

$$\frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta}\ \textsc{LeftNot} \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\ \textsc{RightNot}$$

$$\frac{\Gamma, \phi[x := t] \vdash \Delta}{\Gamma, \forall x.\phi \vdash \Delta}\ \textsc{LeftForall} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \forall x.\phi, \Delta}\ \textsc{RightForall}$$

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma, \exists x.\phi \vdash \Delta}\ \textsc{LeftExists} \qquad \frac{\Gamma \vdash \phi[x := t], \Delta}{\Gamma \vdash \exists x.\phi, \Delta}\ \textsc{RightExists}$$

Figure 3.3: Deduction rules allowed by Lisa's kernel, part 1. Different occurrences of the same symbols need not represent equal elements, but only elements with the same $\mathrm{F(OL)}^2$ normal form. In rules RightForall and LeftExists, x must not appear free in Γ, Δ.

$$\frac{\Gamma \vdash \phi[x := t], \Delta}{\Gamma \vdash \phi[x := (\varepsilon x.\ \phi)], \Delta}\ \textsc{Epsilon}$$

$$\frac{\Gamma, \exists y \forall x.(x = y) \leftrightarrow \phi \vdash \Delta}{\Gamma, \exists! x.\phi \vdash \Delta}\ \textsc{LeftExistsOne}$$

$$\frac{\Gamma \vdash \exists y \forall x.(x = y) \leftrightarrow \phi, \Delta}{\Gamma \vdash \exists! x.\phi, \Delta}\ \textsc{RightExistsOne}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma[(x : A) := (e : A)] \vdash \Delta[(x : A) := (e : A)]}\ \textsc{InstSchema}$$

$$\frac{\Gamma, \phi[f := s] \vdash \Delta}{\Gamma, \forall \vec{x}.s(\vec{x}) \equiv t(\vec{x}), \phi[f := t] \vdash \Delta}\ \textsc{LeftSubst}$$

$$\frac{\Gamma \vdash \phi[f := s], \Delta}{\Gamma, \forall \vec{x}.s(\vec{x}) \equiv t(\vec{x}) \vdash \phi[f := t], \Delta}\ \textsc{RightSubst}$$

$$\frac{}{\vdash t = t}\ \textsc{RightRefl}$$

$$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2}\ \textsc{Restate if } (\bigwedge \Gamma_1 \to \bigvee \Delta_1) \sim_{F(OL)^2} (\bigwedge \Gamma_2 \to \bigvee \Delta_2)$$

$$\frac{}{\Gamma_2 \vdash \Delta_2}\ \textsc{RestateTrue if } \mathrm{True} \sim_{F(OL)^2} (\bigwedge \Gamma_2 \to \bigvee \Delta_2)$$

$$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2}\ \textsc{Weakening if } (\bigwedge \Gamma_1 \to \bigvee \Delta_1) \leq_{F(OL)^2} (\bigwedge \Gamma_2 \to \bigvee \Delta_2)$$

$$\frac{}{\Gamma \vdash \Delta}\ \textsc{Sorry admits a statement without proof. Usage transitively tracked.}$$

Figure 3.4: Deduction rules allowed by Lisa's kernel, part 2.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\phi \vdash \phi}\ \textsc{Hyp}}{\phi \vdash \phi, \psi}\ \textsc{Weakening}}{\vdash \phi, (\phi \to \psi)}\ \textsc{RightImplies} \qquad \dfrac{}{\phi \vdash \phi}\ \textsc{Hyp}}{(\phi \to \psi) \to \phi \vdash \phi}\ \textsc{LeftImplies}}{\vdash ((\phi \to \psi) \to \phi) \to \phi}\ \textsc{RightImplies}$$

Figure 3.5: A proof of Peirce's law in Sequent Calculus. The bottommost sequent (root) is the conclusion.

0 `Hypothesis`	$\phi \vdash \phi$
1 `Weakening`(0)	$\phi \vdash \phi, \psi$
2 `RightImplies`(1)	$\vdash \phi, (\phi \to \psi)$
3 `LeftImplies`(2, 0)	$(\phi \to \psi) \to \phi \vdash \phi$
4 `RightImplies`(3)	$\vdash ((\phi \to \psi) \to \phi) \to \phi$

Figure 3.6: The proof of Peirce's Law as a sequence of steps using classical sequent calculus rules.

−1 `Imported Axiom`	$\vdash \neg(x \in \emptyset)$
0 `Restate`(−1)	$(x \in \emptyset) \vdash$
1 `LeftSubst`(0)	$(x \in y), y = \emptyset \vdash$
2 `Restate`(1)	$(x \in y) \vdash \neg(y = \emptyset)$

Figure 3.7: A proof that if $x \in y$, then $\neg(y = \emptyset)$, using the empty set axiom. x and y are free variables.

Each proof step refers to its premises using numbers, which indicate the place of the premise in the proof. A proof step can also be referred to by multiple subsequent proof steps, so that proofs are actually directed acyclic graphs (DAG) rather than trees. For the proof to be the linearization of a DAG, the proof steps must only refer to numbers smaller than their own as premises in the proof. Figure 3.6 shows the proof of Peirce's Law as linearized in Lisa's kernel. Note however that thanks to the $F(OL)^2$ equivalence checker, Peirce's law can be proven in a single step:

0 `RestateTrue` $\vdash ((\phi \to \psi) \to \phi) \to \phi$.

Moreover, proofs are conditional: they can carry an explicit set of assumed sequents, named "`imports`", which can be used as leaves in the proof. Typically, these imports will contain previously proven theorems, definitions or axioms (see Subsection 3.3.5). For a proof step to refer to an imported sequent, one uses negative integers. −1 corresponds to the first sequent of the import list of the proof, −2 to the second, etc.

Formally, a proof is a pair made of a list of proof steps and a list of sequents:

```
case class SCProof(steps: Seq[SCProofStep], imports: Seq[Sequent])
```

We call the bottom-most sequent of the last proof step of the proof the "conclusion" of the proof. Figure 3.7 shows a proof using an import. Finally, Figure 3.8 shows a proof with quantifiers.

0 `RestateTrue`	$P(x), Q(x) \vdash P(x) \land Q(x)$
1 `LeftForall`(0)	$P(x), \forall(x, Q(x)) \vdash P(x) \land Q(x)$
2 `LeftForall`(1)	$\forall(x, P(x)), \forall(x, Q(x)) \vdash P(x) \land Q(x)$
3 `RightForall`(2)	$\forall(x, P(x)), \forall(x, Q(x)) \vdash \forall(x, P(x) \land Q(x))$
4 `Restate`(3)	$\forall(x, P(x)) \land \forall(x, Q(x)) \vdash \forall(x, P(x) \land Q(x))$

Figure 3.8: A proof showing that $\forall$ factorizes over conjunction.

Subproofs To organize proofs, Lisa's kernel also defines the Subproof proof step. A Subproof is a single proof step in a large proof with arbitrarily many premises:

```
case class SCSubproof(sp: SCProof, premises: Seq[Int]) extends SCProofStep
```

The first argument contains a sequent calculus proof, with one conclusion and arbitrarily many *imports*. The second argument must justify all the imports of the inner proof with previous steps of the outer proof. A Subproof only has an organizational purpose and allows one to more easily write tactics (see Subsection 3.4.3). In particular, the numbering of proof steps in the inner proof is independent of the location of the subproof step in the outer proof.

Sorry Lisa's Kernel includes a step called Sorry, used to represent unimplemented proofs. The conclusion of a Sorry step will always be accepted by the proof checker. Any theorem relying on a Sorry step is not guaranteed to be correct. The usage is, however, transitively tracked, and a theorem relying on the Sorry step is marked as such.

3.3.4 Proof checker

In Lisa, a proof object by itself is not guaranteed to be correct. It is possible to write an incorrect proof. Lisa contains a *proof checking* function, which, given a proof, will verify whether it is correct. To be correct, a proof must satisfy the following conditions:

1. No proof step must refer to itself or a posterior proof step as a premise.
2. Every proof step must be correctly constructed, with the bottom sequent correctly following from the premises by the deduction rule and its arguments.

Given some proof p, the proof checker will verify these points. For most proof steps, this typically involves verifying that the premises and the conclusion match according to a transformation specific to the deduction rule.

Hence, most of the proof checker's work consists of verifying that some formulas, or subformulas thereof, are identical. This is where the equivalence checker comes into play. By checking equivalence rather than strict syntactic equality, a lot of steps

become redundant and can be merged. That way, any number of consecutive LeftAnd, RightOr, LeftNot, RightNot, LeftImplies, RightImplies, LeftIff, LeftRefl, RightRefl, LeftExistsOne and RightExistsOne proof steps can always be replaced by a single Weakening rule. This gives some intuition about how useful the equivalence checker is to simplify proof length.

Depending on whether the proof is correct or incorrect, the proof checking function will output a *judgement* which can be valid or invalid:

```
class SCValidProof(proof: SCProof)
class SCInvalidProof(proof: SCProof, path: Seq[Int], message: String)
```

`SCInvalidProof` indicates an erroneous proof. The second argument points to the faulty proof step (through subproofs, if any), and the third argument is an error message hinting at why the step is incorrectly applied.

3.3.5 Theorems and theories

Theorems don't exist in isolation, but instead depend on some agreed-upon set of axioms, definitions and previously proven theorems. Formally, theorems are developed within theories. A theory is defined by a language, which contains the symbols allowed in the theory, and by a set of axioms, which are assumed to hold true within it.

In Lisa, a `Theory` is a mutable object that starts as the pure theory of predicate logic: It has no known symbols and no axioms. Then we can introduce into it elements of set theory (symbols $\in$, $\emptyset$, $\bigcup$ and set theory axioms, see Subsection 3.4.4) or of any other theory.

To conduct a proof inside a `Theory` and use its axioms, definitions or previously proven theorems, the corresponding statement must be specified in the imports of the proof. Then, the proof can be given to the `Theory` to check, along with *justifications* for all imports of the proof. A justification is either an axiom, a previously proven theorem or a definition. The `Theory` object will check that every import of the proof is properly justified by a *justification* in the theory, i.e., that the proof is in fact not conditional in the theory. Then, it will pass the proof to the proof checker. If the proof is correct, it will return a `Theorem` encapsulating the sequent. This theorem will be allowed to be used in all further proofs as an import, exactly like an axiom.

3.3.6 Definitions

The user can also introduce definitions in the `Theory`. A definition in Lisa's kernel contains a new symbol and the expression it stands for:

```
case class Definition (cst: Constant, expression: Expression)
```

For a definition of a new symbol $c := e$ with $e : A_1 \to A_2 \to ... \to A_n \to B$, the statement introduced by the definition is:

$$c\ x_1\ x_2\ ...\ x_n \equiv e\ x_1\ x_2\ ...\ x_n$$

where $x_1, x_2, \ldots, x_n$ are arbitrary variables of type $A_1, A_2, \ldots, A_n$ respectively, fresh. Note that for the definition to be valid, e may not contain any free variables, all constant symbols in e must be previously defined in the theory, and c must not already be part of the `Theory`. Once a definition has been introduced, future theorems can refer to those definitional axioms by importing the corresponding sequents in their proof and providing justification for those imports when the proof is verified, just like with axioms and theorems.

3.4 Designing a proof assistant, part 2: Lisa's interface

Lisa's kernel offers only the features necessary to write and verify proofs and build mathematical theories, with few concessions to convenience and user-friendliness. Writing proofs directly in the kernel is rather cumbersome. To develop and maintain a mathematical library, Lisa offers a dedicated interface and DSL to write proofs, leveraging some of Scala 3's features.

Listing 3.1 shows an example of what a theory file looks like using Lisa's DSL. In the first line, we declare a new object (or module) to contain our definitions and theorems. On lines 2 and 3, we declare that `x` and `y` are variables of type `Ind` (that is, regular first-order variables). On line 5, we declare a theorem whose statement is that the empty set is a subset of `x`. Since `x` is a free variable, it effectively stands for any expression, so the theorem states that the empty set is a subset of any set. Lines 8–11 contain the three steps of the proof, of the form

```
1   have(`statement`) by `tactic`
```

`` `statement` `` is always a formula or a sequent that we claim is true and which we use as an intermediate toward the goal. `` `tactic` `` is a proof-producing function, possibly taking additional arguments. At lines 8 and 10, we assign a name to the proof steps so that we can refer to them later. `Tautology` and `RightForall` are two tactics provided by Lisa's standard library. `subsetAxiom of (x := ∅, y := x)` is a set-theoretic axiom that we use as a premise of the proof, with a specific instantiation of its free variables.

In this section, we explain in detail the features and implementation of Lisa's DSL.

Acknowledgement of contributions While the design and implementation of Lisa is my own work, several tactics, a large part of the standard library and a wide range of improvements must be attributed to Sankalp Gambhir. Additionally, Dario Halilovic rewrote most of the standard library to use λFOL and the interface described here. Many other people, especially students, made smaller but still important contributions

Listing 3.1: Example of a theory file in Lisa

```
1  object MyTheoryName extends lisa.Main:
2    val x = variable[Ind]
3    val y = variable[Ind]
4
5    val emptySetIsASubset = Theorem(
6      ∅ ⊆ x
7    ) {
8      val s = have((y ∈ ∅) ⟹ (y ∈ x)) by
9        Tautology.from(emptySetAxiom of (x := y))
10     val s2 = have(∀(y, (y ∈ ∅) ⟹ (y ∈ x))) by RightForall(s)
11     have(thesis) by Tautology.from(subsetAxiom of (x := ∅, y := x), s2)
12   }
```

to Lisa's codebase and library, though these contributions are not directly reflected in this section.

3.4.1 Richer syntax

The syntax of Prooflib is similar to the syntax of Lisa's kernel, but the *Sorts*, such as Ind and Prop, are reflected in Scala's type system, so that well-sortedness is checked at compile time, offering more detailed documentation and features. Prooflib's syntax also supports custom printing, such as infix notation, special handling for binders, and more.

Definition 3.4.1 (Sorts)**.**

```
trait Ind
trait Prop
infix trait >>:[I, O]
```

Definition 3.4.2 (Expressions)**.** Expressions in Prooflib always correspond to an underlying expression in Lisa's kernel, which can be accessed using `myExpr.underlying`. Expressions are always *sorted*, and this sort reflects in their Scala type.

```
trait Expr[S]:
  def underlying: kernel.Expr
  def sort: kernel.Sort = underlying.sort
case class Variable[S](name: String) extends Expr[S]
case class Constant[S](name: String) extends Expr[S]
case class App[S, T](f: Expr[S >>: T], arg: Expr[S]) extends Expr[T]
case class Abs[S, T](v: Variable[S], body: Expr[T]) extends Expr[S >>: T]
```

We typically create variables and constants using helper functions, as in:

Example 3.4.3.

```
val x = variable[Ind] //the name "x" is used automatically
val c = variable[Ind]
val ∈ = constant[Ind >>: Ind >>: Prop]
val f = variable[(Ind >>: Prop) >>: Ind]

x ∈ c : Expr[Prop]
val e : Expr[Ind >>: Prop] = λ(x, x ∈ c) //we can use unicode
f(e) : Expr[Ind]
```

Constants can be given custom printing notation. Expressions also support substitutions.

Definition 3.4.4 (Substitution)**.**
Substitution should most often be performed with the `SubstPair` construct, which guarantees well-sortedness:

```
trait SubstPair extends Product:
  type S
  val _: Variable[S]
  val _2: Expr[S]

(x := f(∅)) : SubstPair
g(x, y).subst(x := f(∅), y := x) // == g(f(∅), x)
```

but can also be performed unsafely when sorts are not necessarily known at compile time:

```
//if ill-sorted, may fail.
myExpr.substituteUnsafe(Map(x → s, y → t))

//with sanity runtime check for well-sortedness
myExpr.substituteWithCheck(Map(x → s, y → t))
```

Binders are a particular kind of constant that can be constructed with special syntax, such as:

```
∀ : Expr[(Ind >>: Prop) >>: Prop]
∀(x, x ∈ c) : Expr[Prop] // == App(∀, λ(x, x ∈ c))
```

Finally, expressions build into sequents, which again have an underlying sequent in the kernel.

Definition 3.4.5 (Sequents)**.** Sequents are formally pairs of sets of `Expr[Prop]`.

```
case class Sequent(left: Set[Expr[Prop]], right: Set[Expr[Prop]])
```

Sequents can be built from formulas and collections of formulas:

```
val s = (x ∈ c) ⊢ (f(x) ∈ f(c))
val s2 = () ⊢ (f(x) ∈ f(c))
val s3 = (x ∈ c) ⊢ ()
val s4 = Set(x ∈ c, y ∈ c) ⊢ Set(x = y, x ∈ y)
val s5 = assumptions ⊢ (x = c)
```

The logical semantics of sequents is the same as in the kernel, i.e. a sequent is valid if and only if the conjunction of its left side implies the disjunction of its right side. But it is usually discouraged to have multiple formulas on the right side of a sequent in theorems and lemmas, as it is harder to understand. Using multiple formulas on the right side of a sequent is however allowed in intermediate steps of a proof and in proof tactics.

Sequents, like expressions, support substitutions:

```
val s = Sequent(Set(x ∈ c), Set(f(x) ∈ f(c)))
s.substitute(x := g(∅))
  // == Sequent(Set(g(∅) ∈ c), Set(f(g(∅)) ∈ f(c)))
```

3.4.2 Library

A `library` is an object that corresponds to a particular mathematical and kernel *theory*, containing a set of axioms, definitions and theorems, but also handling error reporting, organization of theorems into files and sections, pretty printing of proofs, options (such as draft mode and caching) and other utilities. Lisa has one main library, but users can also define their own. To use the main library, one needs to create an object that inherits from the `Lisa.Main` object:

```
object MyTheoryName extends lisa.Main:
```

In such an object, one can introduce new definitions and theorems.

Definition 3.4.6 (Definitions)**.**
A definition in Prooflib has the form:

```
val singleton = DEF(λ(x, unorderedPair(x, x)))
```

This has two effects: First, the `DEF` construct registers a new symbol `singleton` in the `library` and the kernel `theory`, with the given definition. Second, this defines a new Scala symbol of the same name, referring to the object `Constant[Ind >>: Ind]("singleton")`, which can directly be used in the current file as well as in future files that import this one. The runtime name "singleton" is automatically fetched from the Scala variable name. The statement resulting from the definition is `singleton(x) ⟛ unorderedPair(x, x)` and can be obtained using `singleton.definition`.

Definition 3.4.7 (Theorems)**.**
A theorem in Prooflib has the form:

```
val membership = Theorem(
  y ∈ singleton(x) ⟺ (y ⟛ x)
) {
  ... //proof script
}
```

Using this construction opens a proof builder in the scope of the proof that helps write proof steps, store current steps, goal and tactics used. The proof builder contains a (mutable) list of proof steps. Each declaration `have(` `statement` `) by` `tactic` adds a new proof step. At the end of the proof script, the `Theorem` construct checks that the last proof step corresponds to the theorem statement and handles the kernel proofs produced by the proof builder and the tactics to obtain an actual `Theorem` object, which is registered in the library and the kernel theory.

Writing a proof in Lisa consists of writing a sequence of proof steps, typically:

```
have(`statement`) by `tactic`
```

`` `statement` `` is simply a formula or sequent, and `` `tactic` `` is a Scala function of type `Sequent ⇒ ProofTacticJudgement`, where a `ProofTacticJudgement` is either a valid

proof step containing a kernel proof, or an error message. In particular, a tactic may fail, which is useful for creating new tactics by combining or repeating existing ones. Tactics may also take additional arguments, and in particular premises. Premises must be known facts: Either an external fact (axioms, theorems and definitions), a proof step already added to the proof, or a fact with a particular instantiation of its free variables using the keyword `of` and a list of substitution pairs:

```
class Proof( ... ):
  type Fact = ExternalFact | ProofStep | InstantiatedFact
  extension (f: Fact):
    def of(substitutions: SubstPair*): InstantiatedFact = ...
```

External facts are automatically imported in the resulting kernel proof.

If a tactic takes a single premise, we can use the `thenHave` construct to chain proof steps without needing to assign them a name:

```
  have(c ≡≡≡ d) by myFavouriteTactic
  thenHave(f(c) ≡≡≡ f(d)) by Congruence
  //equivalent to:
  val step = have(c ≡≡≡ d) by myFavouriteTactic
  have(f(c) ≡≡≡ f(d)) by Congruence.from(step)
```

Alternatively, the `laststep` keyword can be used to refer to the last proof step of the current proof, which is useful if the second step takes more than one premise. Some tactics may also take a variable number of premises. Subsection 3.4.3 describes some of Lisa's existing tactics and how to write new ones.

Subproofs. Additionally, it is possible to organize a proof into subproofs, using

```
  have(`statement`) subproof {
    ... //subproof steps
  }
```

which opens a new proof builder for the subproof. In addition to external facts and previous proof steps of the subproof, statements of parent proofs also count as `Facts`.

```
class Subproof(parent: Proof) extends Proof( ... ):
  type ExternalFact = parent.Fact
```

Here `parent.Fact` is a path-dependent type, and steps of a subproof are not facts in a different subproof, even if the step leaks outside the scope of its subproof. Note that this is checked at compile time.

In a proof, we can also use the `assume` function, which adds a formula as an implicit assumption to all the future steps in the proof. Subproofs inherit the assumptions of their parent proofs, but assumptions do not propagate back to parent proofs. Altogether, these features allow one to write a proof such as in Listing 3.2.

Finally, if the proof is correct, we can run the theory file (its main function is inherited from `lisa.Main`) and we obtain the output of Listing 3.3. If the proof was incorrect, the theorem would be shown in red.

Listing 3.2: Proof of the extensionality of singleton sets in lisa's standard library

```
val extensionality = Theorem(
    (singleton(x) ≡ singleton(y)) ⟺ (x ≡ y)
  ) {
  val s = have(singleton(x) ≡ singleton(y) ⊢ (x ≡ y)) subproof {
    assume(singleton(x) ≡ singleton(y))
    have(x ∈ singleton(x) ⟺ x ∈ singleton(y)) by Congruence
    thenHave(thesis) by Tautology.from(
      laststep,
      Singleton.membership of (x := x, y := x),
      Singleton.membership of (x := y, y := x)
    )
  }
  val s2 = have((x ≡ y) ⊢ (singleton(x) ≡ singleton(y))) by Congruence
  have(thesis) by Tautology.from(s, s2)
  }
```

Listing 3.3: Result of running the file `Singleton.scala` of lisa's standard library

```
Definition of singleton(x) := {x, x}
Theorem membership :=  ⊢ y ∈ {x} ⟺ y = x
Theorem nonEmpty :=  ⊢ ¬({x} = ∅)
Theorem extensionality :=  ⊢ {x} = {y} ⟺ x = y
Theorem equalsUnorderedPair :=  ⊢ {x} = {y, z} ⟺ x = y ∧ x = z
```

Listing 3.4: Propositional solver as Lisa tactic

```
1  def tauto(using proof: Proof)
2          (_f: Expr[Prop]): proof.ProofTacticJudgement = TacticSubproof {
3    val f = normalForm(_f)     //computes OL-normal form
4    if f == ⊤ then have(f) by Hypothesis
5    else
6      val a = findBestAtom(f)
7
8      have(tauto(f.substitute(a, ⊤)))
9      val step = thenHave(a ⊢ f) by Substitute(⊤ ⟺ a)
10     have(tauto(f.substitute(a, ⊥)))
11     val step2 = thenHave(!a ⊢ f) by Substitute(⊥ ⟺ a)
12
13     have(f) by Cut(step, step2)
14 }
```

3.4.3 Tactics

At its core, a Lisa tactic is simply a Scala function that returns a kernel proof. Tactics can be combined, repeated and can mix together the Lisa proof DSL, Scala libraries and arbitrary programming constructs. This makes it possible to write tactics, even complex ones, easily, in the same files and language as the rest of the proof. To illustrate this, Listing 3.4 shows how to implement a simplified version of Lisa's `Tautology` tactic (solving propositional tautologies), following a simple DPLL-based approach.

Lines 1–2 define the `tauto` function, which takes as input the current proof and a formula to prove. The `using` keyword indicates that the argument is passed implicitly. The function returns a `ProofTacticJudgement` through the `TacticSubproof` utility. This is boilerplate that allows one to more precisely report errors in tactics and in their use. Line 3 computes the *OL*-normal form of the formula, using an auxiliary function `normalForm` (not shown here). Note that this does not require a dedicated proof step! Line 4 checks if the formula is $\top$, in which case it adds a proof step using the `Hypothesis` tactic. Line 6 picks an atom in the formula, using an auxiliary function `findBestAtom`. Lines 8–11 recursively call `tauto` on the formula where the chosen atom is substituted by $\top$ or $\bot$, to show that the formula `f` holds both assuming `a` and assuming `!a`. Finally, line 13 combines the two proof steps using the `Cut` tactic to conclude that `f` holds unconditionally.

This is a minimal implementation, but it can be extended in various ways. We could implement different heuristics to choose the best atom (a reasonable choice may be the atom that appears most often in the formula), add caching to avoid recomputing proofs of the same formula multiple times, or even parallelize the two recursive calls to `tauto`. To improve error reporting, we can for example test if the recursive calls to `tauto` succeeded:

Tactic	Description
`Restate`	Accepts any transformation supported by Lisa's equivalence checker.
`Tautology`	Decision procedure for propositional logic.
`Congruence`	Uses an e-graph to solve systems of equalities and equivalences.
`Substitute`	Under an assumption of fact $a = b$, substitute a by b in a known fact.
`Tableaux`	Proof-producing tableaux solver for first-order logic.
`Goeland`	Calls the Goéland ATP and reconstructs the SC-TPTP proof.
`Prover9`	Calls the Prover9 ATP and reconstructs the SC-TPTP proof.
`Egg`	Calls the egg e-graph tool and reconstructs the SC-TPTP proof.

Table 3.2: Common tactics provided by Lisa's standard library.

```
  ...
else if f == ⊥ then return proof.InvalidProofTactic("Not a Tautology")
  ...
  val recStep = tauto(f.substitute(a, ⊤))
  val step = if recStep.isValid then
    have(a ⊢ f) by Substitute(⊤ ⟺ a)(have(recStep))
  else
    return proof.InvalidProofTactic(s"Failed to prove $f under $a")
  ...
```

Table 3.2 contains a non-exhaustive list of common tactics provided by Lisa's standard library, on top of tactics resulting from the base proof steps of Figure 3.4.

3.4.4 Lisa's standard library

Lisa's standard library is based on set theory, and more precisely ZFC axioms. ZFC stands for Zermelo-Fraenkel with Choice, and is the most generally accepted foundation of mathematics in the mathematical community. Lisa's definition of ZFC contains the symbols:

```
∈ : Constant[Ind >>: Ind >>: Prop] //membership
⊆ : Constant[Ind >>: Ind >>: Prop]  //subset
∅ : Constant[Ind]                    //empty set
unorderedPair : Constant[Ind >>: Ind >>: Ind] //unordered pair
⋃ : Constant[Ind >>: Ind]            //union
𝒫 : Constant[Ind >>: Ind]   //power set
```

as well as the axioms in Listing 3.5. There is an additional axiom, Tarski's axiom, which implies the existence of large cardinals and "universes". Those universes are closed under the construction of ZFC, and will in the future allow Lisa to simulate theorems from the Calculus of Inductive Constructions (CIC) with universes, as used in Rocq and Lean. For now, Tarski's axiom has not been used, and since its definition is complex, it is not shown here.

One of the very first developments of Lisa's library is to define and prove properties about set comprehension. Comprehension illustrates the con-

Listing 3.5: Axioms of Lisa's standard library

```
val extensionalityAxiom =
  Axiom(∀(z, z ∈ x ⟺ z ∈ y) ⟺ (x ⩵ y))

val pairAxiom =
  Axiom(z ∈ unorderedPair(x, y) ⟺ (z ⩵ x) ∨ (z ⩵ y))

val comprehensionSchema =
  Axiom(∃(z, ∀(x, x ∈ z ⟺ (x ∈ y) ∧ φ(x))))

val emptySetAxiom =
  Axiom(x ∉ ∅)

val unionAxiom =
  Axiom(z ∈ ⋃(x) ⟺ ∃(y, (y ∈ x) ∧ (z ∈ y)))

val subsetAxiom =
  Axiom((x ⊆ y) ⟺ ∀(z, (z ∈ x) ⟹ (z ∈ y)))

val powerSetAxiom =
  Axiom(x ∈ 𝒫(y) ⟺ x ⊆ y)

val infinityAxiom =
  Axiom(∃(x, ∅ ∈ x ∧ ∀(y, (y ∈ x) ⟹
    ⋃(unorderedPair(y, unorderedPair(y, y))) ∈ x)))

val axiomOfFoundation =
  Axiom(x ≠ ∅ ⟹ ∃(y ∈ x, ∀(z, z ∈ x ⟹ z ∉ y)))

val replacementSchema =
  Axiom(∀(x ∈ A, ∀(y, ∀(z, P(x)(y) ∧ P(x)(z) ⟹ (y ⩵ z)))) ⟹
    ∃(B, ∀(y, y ∈ B ⟺ ∃(x ∈ A, P(x)(y)))))
)
```

venience of the λFOL syntax, as well as the expressiveness of Lisa's DSL:

```
val setComprehension: Constant[Ind >>: (Ind >>: Prop) >>: Ind] =
  DEF(λ(y, λ(φ, ε(z, ∀(x, x ∈ z ⟺ x ∈ y ∧ φ(x))))))

setComprehension.printAs( ... )

extension (e: Expr[Prop]) {
  def |(φ: Expr[Prop]): Expr[Ind] = e match {
    //{x ∈ y | φ(x)}
    case (x: Variable[Ind]) ∈ y ⇒
      setComprehension(y)(λ(x, φ))
    case _ ⇒ ...
  }
}
```

This enables a natural mathematical syntax[1], for example:

```
val membership = Theorem(
  x ∈ { x ∈ y | φ(x) } ⟺ x ∈ y ∧ φ(x)
)
```

Replacement is similarly developed and given its own notation, as in:

```
val membership = Theorem(
  y ∈ { F(x) | x ∈ A } ⟺ ∃(x ∈ A, F(x) ⩦ y)
)
```

Note that the two theorems above are members of different objects, so that each can be used as `Comprehension.membership` and `Replacement.membership` respectively. This style of encapsulating theorems about a particular construction in modules whose names reflect the construction is used throughout Lisa's standard library.

The library then formalizes further set-theoretic constructions and useful theorems about them, largely following the first chapter of [90]. More precisely, the following constructions are defined and their basic properties proven:

- Singleton sets, set intersection, difference
- Ordered pairs, defined as Kuratowski pairs ($(a, b) := \{\{a\}, \{a, b\}\}$) with accessor functions `first` and `second` and Cartesian products
- Relations with various properties (reflexive, symmetric, transitive, antisymmetric, total, ...) and operations (closure, complement, composition, ...)
- Functions (`functionBetween(f)(A)(B)`) as particular relations that are functional and total, with properties (surjectivity, injectivity) and operations (composition, restriction and union of functions). Lisa also defines function spaces (`A → B`) and applications (`f @ x`), with properties necessary for type checking and computation such as:

[1] Remember that this is still valid Scala code, even if it doesn't look like it!

```
val appTyping = Theorem(
  (f ∈ A → B, x ∈ A) ⊢ ((f @ x) ∈ B)
)
```

We also define a function constructor `fun` as described in Definition 3.2.4:

```
val fun = DEF(λ(A, λ(F, { (x, F(x)) | x ∈ A })))
```

`fun` is given its own custom notation, so that we can for example write:

```
val identityTyping = Theorem(
  fun(x ∈ A, x) ∈ A → A
)
```

- Lisa then formalizes orders, order isomorphisms, total orders and in particular well-orders
- Ultimately Lisa's standard library formalizes ordinal numbers, with all necessary properties leading to the transfinite induction and recursion theorems:

```
val transfiniteInduction = Theorem(
  ∀(α, ordinal(α) ⟹ ∀(β ∈ α, P(β)) ⟹ P(α)) ⊢ ordinal(α) ⟹ P(α)
)
val transfiniteRecursion = Theorem(
  ordinal(α) ⊢ ∃(G, ∀(β ∈ α, G(β) ⩦ F(β)(G ↾ β)))
)
```

3.5 Embedding HOL in set theory with λFOST

In Section 3.1 and Section 3.2, we have developed the theory of functions and simple types in set theory, which form the basis of Higher Order Logic (HOL). In the present section, we will show how we can formally embed HOL in set theory. Such an embedding will not only provide us with additional ways to reason about set-theoretic functions, but ultimately will permit the import of theorems and proofs from proof assistants based on HOL, such as Isabelle/HOL, HOL4 or HOL Light into Lisa or a system with similar foundations.

It is well known and easy to show that set theory axioms provide a model for HOL, meaning that every function and element of the universe of HOL can be interpreted as some set. In particular in this model, HOL functions are mapped to set-theoretic functions, and HOL types are mapped to the corresponding sets. However, this semantic embedding is of little practical use; what we need is to translate the syntax of HOL to the syntax of (first-order) set theory. With pure first-order logic, this is quite difficult precisely because there is no way to directly represent λ-expressions. λFOL, on the other hand, can represent such terms easily, and hence the translation is much simpler. In fact, this was one of the original motivations for the development of λFOL.

Acknowledgement of contributions All the contributions of this section are the result of joint work with Sankalp Gambhir.

3.5.1 From higher-order logic to set theory

There exist several variations of higher-order logic (HOL). We consider it as it is defined in HOL Light. Its language is the simply typed λ-calculus with top-level polymorphism (or the Hindley–Milner type system). Each variable in an HOL term is associated with a single type, which may contain type variables. We denote by V^λ the set of HOL variables and by T^λ the set of type variable symbols of HOL. The deduction rules of HOL Light are described in Figure 3.10 [2].

We wish to define an embedding $(\!|\cdot|\!)$ from HOL sequents to FOL sequents such that if an HOL sequent s is provable in HOL, then $(\!|s|\!)$ is provable in λFOST. Moreover, we would like the embedding to be as natural as possible: HOL functions should be mapped to set-theoretic functions, and HOL types should be mapped to the corresponding set. First, we need to define equality on a type A as a set-theoretic function.

Definition 3.5.1. Let $\mathbb{B}$ be the set $\{\text{true}, \text{false}\}$. Define the (curried) equality function on a type A as

$$\mathcal{E}\ A := \varepsilon f \in A \Rightarrow (A \Rightarrow \mathbb{B}).\ \forall x, y \in A.\ f@x@y \equiv \top \Leftrightarrow x \equiv y$$

[2] HOL Light also admits a choice function and an infinity axiom later in the library development, which are justified in ZFC by the choice axiom and infinity axiom. These two additional axioms are largely tangential to the concerns of the present work. While ETA is also formally an axiom in HOL Light, we consider it as a basic rule because set-theoretic functions are naturally extensional.

Note that, by definition, for all A, $(\mathcal{E}\ A) \in (A \Rightarrow (A \Rightarrow \mathbb{B}))$. Moreover,

$$x \in A, y \in A \vdash ((\mathcal{E}\ A)\ x\ y \equiv \top) \Leftrightarrow (x \equiv y) \tag{3.9}$$

We use the notation $s \sim_A t$ to denote $(\mathcal{E}\ A)@s@t$.

The embedding of HOL terms is defined recursively as follows: We denote by $\mathcal{B}$ the type of Booleans containing two elements, by i the infinite type of individuals asserted by HOL axioms, and by $=_A$ the built-in polymorphic function representing equality on type A.

Definition 3.5.2 (Embedding of HOL into λFOST)**.**

$$\begin{aligned}
(\!|X|\!) &= X \\
(\!|T_1^\lambda \to T_2^\lambda|\!) &= (\!|T_1^\lambda|\!) \Rightarrow (\!|T_2^\lambda|\!) \\
(\!|\mathcal{B}|\!) &= \mathbb{B} \\
(\!|i|\!) &= \mathbb{N} \\
(\!|x : A|\!) &= x \\
(\!|=_A \colon A \to A \to \mathbb{B}|\!) &= \mathcal{E}((\!|A|\!)) \\
(\!|(f : A \to B)(t : A) : B|\!) &= (\!|f : A \to B|\!)@(\!|t : A|\!) \\
(\!|\lambda x : A.\ t : B : A \to B|\!) &= \lambda x \in (\!|A|\!).\ (\!|t|\!)
\end{aligned}$$

where $T_1, T_2 \in T^\lambda$ are types in HOL and $X \in V^\lambda$ is a variable symbol.

There is one small issue with this encoding, which is that we are losing type information associated with variables, as well as the HOL assumption that type variables cannot represent empty types. We will need a *non-emptiness context*, to handle type variables and a *typing context*, to carry over information regarding types of variables.

The following definition defines ctx^N (non-emptiness) and ctx^T (variable typing). Assume for simplicity and without loss of generality that different HOL variables have different identifiers, so that, for example, $x : A$ and $x : B$ do not appear together in the same proof.

Definition 3.5.3 (Non-emptiness context)**.** The non-emptiness context of an HOL term $\mathrm{ctx}^N(t)$ is the set of assumptions $A \not\equiv \emptyset$ for every type variable A in t. This also includes type variables in the type signature of polymorphic constant symbols.

Definition 3.5.4 (Typing context)**.** The typing context of an HOL term is a set of λFOST formulas of the form $x \in T$ and is computed recursively as follows:

$$\begin{aligned}
\mathrm{ctx}^T(x : T) &= \{x \in T\} \\
\mathrm{ctx}^T(c) &= \emptyset \text{ for } c \text{ a constant symbol} \\
\mathrm{ctx}^T(f\ t) &= \mathrm{ctx}^T(f) \cup \mathrm{ctx}^T(t) \\
\mathrm{ctx}^T(\lambda x : T.\ t) &= \mathrm{ctx}^T(t) \setminus \{x \in T\}
\end{aligned}$$

$$\frac{}{\Gamma, t \in A \vdash t \in A, \Delta}\ \textsc{Var}$$

$$\frac{\Gamma, x \in A \vdash t_x \in B, \Delta}{\Gamma \vdash (\lambda x \in A.\ t_x) \in A \Rightarrow B, \Delta}\ \textsc{Fun} \quad \text{(if } x \text{ is not free in } \Gamma, \Delta)$$

$$\frac{\Gamma \vdash t_1 \in A \Rightarrow B, \Delta \qquad \Sigma \vdash t_2 \in A, \Pi}{\Gamma, \Sigma \vdash t_1 @ t_2 \in B, \Delta, \Pi}\ \textsc{Apply}$$

Figure 3.9: Typing rules for simply typed λ-calculus, in set theory.

The *context*, $\mathrm{ctx}(t)$, of an HOL term t, is $\mathrm{ctx}^N(t) \cup \mathrm{ctx}^T(t)$.

Example 3.5.5. Let $x : X$, $y : Y$, $f : Y \to X$, $g : X \to Y$. We omit type annotations from variables in lambda-terms, except when bound.

$$\begin{array}{lcl}
(\!|\lambda x : X.\ x|\!) & = & \lambda x \in X.\ x \\
\mathrm{ctx}(\lambda x : X.\ x) & = & \{X \neq \emptyset\} \\
\\
(\!|(\lambda x : X.\ y)\ f|\!) & = & (\lambda x \in X.\ y) @ f \\
\mathrm{ctx}((\lambda x : X.\ y)\ f) & = & \{X \neq \emptyset, Y \neq \emptyset, y \in Y, f \in Y \Rightarrow X\} \\
\\
(\!|\lambda y : Y.\ (\lambda x : X.\ y) =_{X \to Y} g|\!) & = & \lambda y : Y.\ (\lambda x \in X.\ y) \sim_{(X \Rightarrow Y)} g \\
\mathrm{ctx}(\lambda y : Y.\ (\lambda x : X.\ y) =_{X \to Y} g) & = & \{X \neq \emptyset, Y \neq \emptyset, g \in X \Rightarrow Y\}
\end{array}$$

We can now define the embedding of sequents:

Definition 3.5.6. Let $s = t_1, \ldots, t_n \vdash t$ be an HOL sequent. Define the embedding $(\!|s|\!)$ as

$$\mathrm{ctx}\,(t_1), \ldots, \mathrm{ctx}\,(t_n), \mathrm{ctx}\,(t), (\!|t_1|\!) = \top, \ldots, (\!|t_n|\!) = \top \quad \vdash \quad (\!|t|\!) = \top$$

3.5.2 Type checking

Equation 3.7 from Theorem 3.2.5 expresses beta-reduction for set-theoretic functions. Applying it recursively allows evaluating expressions built from fun and apply. Unlike the β-reduction of λFOL, which is always performed implicitly by the logical theory, Equation 3.7 is a theorem that needs to be applied as a proof step. Moreover, to reduce an expression $(\lambda x \in A.\ F\ x @ y)$, this theorem is conditional on the well-typedness of the application, that is, $y \in A$. y may itself be built from fun and apply, so we have to produce proofs that such terms are well-typed. Of course, deciding for arbitrary terms if $y \in A$ is undecidable, but in the fragment of fun and apply, which corresponds to simply typed λ-calculus, we can automate the proof. From Equation 3.6, Equation 3.5 and rules of λFOL, we can obtain the inference rules of Figure 3.9, which look just like the usual typing rules from simply typed λ-calculus.

Listing 3.6 shows a simple implementation of proof-producing type checking in Lisa

for simply typed λ-calculus in set theory. The function proves a typing judgement for an expression t by recursively computing and proving the type of its subterms. The Apply rule handles cases where t is of the form $f@arg$, for some terms f and arg; it computes the type of both f and arg with the corresponding proof and then concludes using the inference rule Apply from Figure 3.9.

The Fun rule handles cases where t is of the form $\lambda x \in A.\ t_x$ (which is formally $\operatorname{fun} A\ (\lambda x : T.\ t_x)$). Here we recursively prove under the assumption $x \in A$ that t_x has some type B, which corresponds to the premise of the rule Fun.

Finally, Var handles expressions which are variables, such as x, but also any other term from λFOL and set theory that does not match the two previous cases. This allows us to use arbitrary mathematical expressions as part of the calculus. In these cases, we fetch in the assumptions of the current state whether the term t is already known to have some type, and otherwise the procedure fails.

Listing 3.6: "Type checking in set theory"

```
def typecheck(t:Expr[T]): Expr[T] =
  t match
    case f @ arg ⇒
      val f_type = have(typecheck(f))
      val f_proof = laststep
      val arg_type = have(typecheck(arg))
      val arg_proof = laststep
      assert(arg_type == f_type.in)
      have(t ∈ f_type.out) by Typing.Apply(f_proof, arg_proof)
    case fun(x, typ, body) ⇒
      val body_type = have(Subproof{assume(x ∈ typ); typecheck(body)})
      val body_proof = laststep
      have(t ∈ (x.typ ⇒ body_type)) by Typing.Fun(body_proof, x.typ)
    case _ ⇒ assumptions.collectFirst{
      case (s ∈ typ) if t == s ⇒
        have(t ∈ typ) by Typing.Hyp
        typ
   }
```

Polymorphism The tactic can naturally be extended to support polymorphism. For example, let the polymorphic identity function

$$\operatorname{Id} A := \lambda x \in A.\ x.$$

$\operatorname{Id} A$ has type $A \Rightarrow A$, i.e. $\vdash (\operatorname{Id} A) \in A \Rightarrow A$. This statement can be instantiated to obtain the typing judgement corresponding to any particular instantiation of Id, for example, $\vdash (\operatorname{Id} \mathbb{R}) \in \mathbb{R} \Rightarrow \mathbb{R}$. This property can then be instantiated adequately for any set in place of A. We added support for such top-level polymorphism to the `ProofType` tactic, so that it can automatically type polymorphic constants embedded this way.

$$\frac{}{\vdash t =_A t}\ \texttt{REFL} \qquad \frac{\Gamma \vdash s =_A t \qquad \Delta \vdash t =_A u}{\Gamma, \Delta \vdash s =_A u}\ \texttt{TRANS}$$

$$\frac{\Gamma \vdash s =_{A\to B} t \qquad \Delta \vdash u =_A v}{\Gamma, \Delta \vdash s(u) =_B t(v)}\ \texttt{MK\_COMB} \qquad \frac{}{\vdash \lambda(x : A.t)x =_B t}\ \texttt{BETA}$$

$$\frac{\Gamma \vdash s =_B t}{\Gamma \vdash (\lambda x : A.s) =_{A\to B} \lambda(x : A.t)}\ \texttt{ABS} \qquad \frac{}{p \vdash p}\ \texttt{ASSUME}$$

$$\frac{\Gamma, q \vdash p \qquad \Delta, p \vdash q}{\Gamma, \Delta \vdash p =_{\mathcal{B}} q}\ \texttt{DEDUCT\_ANTISYM\_RULE} \qquad \frac{\Gamma \vdash p =_{\mathcal{B}} q \qquad \Delta \vdash p}{\Gamma, \Delta \vdash q}\ \texttt{EQ\_MP}$$

$$\frac{\Gamma \vdash p}{\Gamma[\vec{x} := \vec{t}] \vdash p[\vec{x} := \vec{t}]}\ \texttt{INST} \qquad \frac{\Gamma \vdash p}{\Gamma[\vec{X} := \vec{A}] \vdash p[\vec{X} := \vec{A}]}\ \texttt{INSTTYPE}$$

$$\frac{}{\vdash (\lambda x.tx) =_A t}\ \texttt{ETA}$$

Figure 3.10: Deduction rules for higher-order logic as implemented in HOL Light.

3.5.3 Simulating HOL proofs

The goal of the section is to demonstrate that HOL Light proof steps can be simulated by proofs in our encoding.

Theorem 3.5.7 (Simulating HOL proofs in λFOST)**.** Let

$$\frac{s_1 \qquad \ldots \qquad s_n}{s}$$

be an instance of a deduction rule of HOL from Figure 3.10. Then

$$\frac{(\!|s_1|\!) \qquad \ldots \qquad (\!|s_n|\!)}{(\!|s|\!)}$$

is admissible in λFOST (rules of sequent calculus and axioms of set theory).

The simulation of a proof step can in general be split into two parts: first, produce a proof under arbitrary typing and context assumptions, and then handle the modifications in context. For example, let $x : \mathcal{B}, f : \mathcal{B} \to \mathcal{B}, g : \mathcal{B} \to \mathcal{B}$ and consider a `TRANS` step deducing

$$\frac{\Gamma \vdash x =_{\mathcal{B}} f\ x \qquad \Gamma \vdash f\ x =_{\mathcal{B}} g\ x}{\Gamma \vdash x =_{\mathcal{B}} g\ x}$$

and let $c_\Gamma = \mathrm{ctx}(\Gamma)$. We wish to obtain a proof of

$$\frac{\begin{array}{ll} x \in \mathbb{B}, f \in \mathbb{B} \Rightarrow \mathbb{B}, & c_\Gamma, (\!|\Gamma|\!) \vdash (x \sim_{\mathbb{B}} f@x) = \top \\ x \in \mathbb{B}, f \in \mathbb{B} \Rightarrow \mathbb{B}, g \in \mathbb{B} \Rightarrow \mathbb{B}, & c_\Gamma, (\!|\Gamma|\!) \vdash (f@x \sim_{\mathbb{B}} g@x) = \top \end{array}}{x \in \mathbb{B}, g \in \mathbb{B} \Rightarrow \mathbb{B}, c_\Gamma, (\!|\Gamma|\!) \vdash (x \sim_{\mathbb{B}} g@x) = \top}$$

This should follow from applying Equation 3.9 to each premise, using transitivity of first-order equality, and applying back Equation 3.9 to the result. However, to apply

this lemma, we need the facts $f\ x \in \mathbb{B}$ and $g\ x \in \mathbb{B}$. We can prove both automatically using our type checking tactic (Listing 3.6). Afterwards, the $f \in \mathbb{B} \Rightarrow \mathbb{B}$ assumption from the premise will stay in the conclusion, yielding:

$$x \in \mathbb{B}, f \in \mathbb{B} \Rightarrow \mathbb{B}, g \in \mathbb{B} \Rightarrow \mathbb{B}, f\ x \in \mathbb{B}, g\ x \in \mathbb{B}, c_\Gamma, (\!|\Gamma|\!) \vdash (x \sim_{\mathbb{B}} g\ x) = \top$$

which is a correct conclusion, but contains too many assumptions. Fortunately, these assumptions can be eliminated.

Eliminating lingering assumptions Pursuing the example above, let $L = \{x \in \mathbb{B}, g \in \mathbb{B} \Rightarrow \mathbb{B}, c_\Gamma, (\!|\Gamma|\!)\}$ and $R = (x \sim_{\mathbb{B}} g\ x) = \top$. We want to simulate the following proof step:

$$\frac{f \in \mathbb{B} \Rightarrow \mathbb{B}, f\ x \in \mathbb{B}, g\ x \in \mathbb{B}, L \vdash R}{L \vdash R}$$

First, we prove (automatically) the non-elementary typing assumptions $(x \in \mathbb{B}, f \in \mathbb{B} \Rightarrow \mathbb{B}) \vdash f\ x \in \mathbb{B}$ by recursing over the structure of $f\ x$ (using Listing 3.6) and similarly for g. Then, note that f is not free anywhere but in its typing assumption: we can quantify it to $\exists f.f \in \mathbb{B} \Rightarrow \mathbb{B}$ using the `leftExists` rule from first-order logic. Now, this statement is provable, as it can be deduced from the non-emptiness of $\mathbb{B}$. Formally, we obtain the following proof:

$$\dfrac{\dfrac{f \in \mathbb{B} \Rightarrow \mathbb{B}, f\ x \in \mathbb{B}, g\ x \in \mathbb{B}, L \vdash R \qquad \dfrac{\dots}{x \in \mathbb{B}, f \in \mathbb{B} \Rightarrow \mathbb{B} \vdash f\ x \in \mathbb{B}}}{f \in \mathbb{B} \Rightarrow \mathbb{B}, g\ x \in \mathbb{B}, L \vdash R}\ \textsc{Cut}}{\vdots}$$

$$\dfrac{\dfrac{\dfrac{f \in \mathbb{B} \Rightarrow \mathbb{B}, L \vdash R}{\exists f.f \in \mathbb{B} \Rightarrow \mathbb{B}, L \vdash R}\ \textsc{LeftExists} \qquad \dfrac{\dots}{\vdash \exists f.f \in \mathbb{B} \Rightarrow \mathbb{B}}}{L \vdash R}\ \textsc{Cut}}{}$$

This example covers statements corresponding to typing context. Non-emptiness context can similarly be eliminated. Note that the non-emptiness of a type variable A is necessary to eliminate a typing assumption of the form $x \in A$. After having applied the proof step, we need to eliminate the assumptions associated with symbols that do not appear in the conclusion. We call these "lingering assumptions" and proceed as follows:

1. Find a variable type assignment $x \in T$. Using LeftExists, generalize to $\exists x.x \in T$. Using the type variable's non-emptiness assumptions, prove that $\exists x.x \in T$ (i.e. T is non-empty). Eliminate $\exists x.x \in T$. Iterate on the next unused variable.

2. Find a non-emptiness assumption $A \neq \emptyset$ for a type variable that does not appear anywhere else. Using LeftExists, generalize to $\exists A.A \neq \emptyset$, which is of course provable without assumption, and eliminate it. Iterate on the next unused type variable.

Simulating HOL steps We briefly hint at how steps of HOL can be simulated in λFOST, leaving implicit concerns regarding proofs of type checking and context elimination, which were addressed above.

- `REFL` is simulated with Equation 3.9 and reflexivity of first-order equality.
- `TRANS` is similarly simulated with Equation 3.9 and transitivity of first-order equality.
- `MK_COMB` is simulated with Equation 3.9 and substitution of equals for equals in first-order logic.
- `ABS` comes from β-equivalence in λFOL.
- `ETA` follows from Equation 3.8.
- `BETA` steps follow from Equation 3.7.
- `ASSUME` is simply a Hypothesis step in sequent calculus.
- `EQ_MP` is simulated with Equation 3.9 and substitution of equals.
- `DEDUCT_ANTISYM_RULE` steps follow from propositional extensionality:

$$p \in \mathbb{B}, q \in \mathbb{B}, (p = \top) \iff (q = \top) \vdash p = q$$

- `INST` follows from instantiation of free variables in first-order logic.
- `INSTTYPE` corresponds to instantiation of free variables.

This concludes our simulation of the various proof tactics in λFOST leading to Theorem 3.5.7.

Corollary 3.5.8. Let s be an HOL sequent. Then a proof of s in HOL can be transformed in a proof of $(\!|s|\!)$ in λFOST.

All HOL Light steps have been implemented in Lisa, and as a result HOL Light-style proofs can be directly written in Lisa. An example is given in Listing 3.7.

We have demonstrated how to embed HOL into conventional first-order logic axiomatization of set theory. Our original published work [43] was not based on λFOL, but rather on standard first-order logic with axiom schemas, which was the foundation of Lisa at the time. In this encoding, it was not possible to represent λ-abstractions directly, for the reasons we described in Section 3.1. Instead, our approach for pure FOL consists in replacing every λ-abstraction appearing in an HOL sequent by a constant symbol c, and adding definition principles stating that c is equal to the corresponding λ-abstraction in the context. This encoding of HOL in pure FOL is significantly more cumbersome, especially when representing HOL proofs. Among other technicalities, it

Listing 3.7: Two simple HOL Light-style proofs in Lisa. Note that * denotes application.

```
val COMPOSE_ID = HOLTheorem(λ(x, λ(y, y) * x) * z =:= z) {
  val s = BETA(λ(x, λ(y, y) * x) * x)
  val s2 = INST(Seq((x, z)), s)
  val s3 = BETA(λ(y, y) * y)
  val s4 = INST(Seq((y, z)), s3)
  _TRANS(s2, s4)
}

val TWO_BETA = HOLTheorem(λ(x, λ(y, λ(x, y)) * x) * w =:= λ(x, w)) {
  val b = BETA(λ(x, λ(y, λ(x, y)) * x) * x)
  val bw = INST(Seq((x, w)), b)
  val b2 = BETA(λ(y, λ(x, y)) * y)
  val b2w = INST(Seq((y, w)), b2)
  _TRANS(bw, b2w)
}
```

requires explicit proofs of alpha-equivalence and maintaining a canonical form for the definition principle in the context that was otherwise not stable under some proof steps.

While this encoding worked reasonably well in practice for terms with few abstractions, representing HOL proofs was very impractical: the size of the first 15 proofs of the HOL Light library was multiplied by a factor of 100 when imported into Lisa, and the implementation of all proof steps was very involved and bug-prone. In contrast, and while automated transfer of the HOL library into Lisa is still undergoing optimization, the current embedding based on λFOL is at least an order of magnitude more efficient.

These results definitely motivated extending the foundation of Lisa to λFOL, which makes the embedding of HOL rather straightforward, as we have seen in this section.

3.6 Interoperability of proof systems with SC-TPTP

Distinct from Interactive Theorem Provers (ITPs) such as Lisa, Automated Theorem Provers (ATPs) aim to solve logical problems with minimal or no human guidance. While ITPs allow developing complex libraries of definitions and theorems, ATPs focus on proving individual conjectures automatically, often using highly optimized search strategies and decision procedures. It would seem that ITPs are a prime use case for ATPs, where they could be used as proof tactics. When developing a proof assistant such as Lisa, developing an entire library from scratch is very time-consuming, requiring one to prove a number of standard lemmas and to implement many decision procedures as tactics for common theories, such as for equality, propositional reasoning or arithmetic. On the other hand, existing ATPs already implement many of these decision procedures with the highest level of optimizations and efficiency. Reimplementing standard decision procedures and search heuristics inside the ITP is often unsatisfactory: large, highly-optimized automated theorem provers (ATPs) and SMT solvers already exist and continuously improve, consuming years of engineering effort that it would be wasteful to duplicate.

However, in practice this typically does not happen due to a number of challenges linked to proof reconstruction: the ITPs cannot simply trust the ATP verdict as an oracle, as this would compromise the ITP's trusted kernel. Instead, a formal proof of the theorem must be found in some way. Unfortunately, ATPs typically do not produce proofs in a format that can be directly checked inside the ITP. For this reason, the main approach to integrating ATPs into ITPs is to use them as so-called "hammers": the ITP invokes the ATP on a query, and uses intermediate steps discovered by the ATP and specific lemmas it used to reconstruct a proof inside the ITP using a less powerful but proof-producing tactic. This approach is mainly employed in the Isabelle proof assistant through the Sledgehammer tool [8], although similar tools exist in other proof assistants [9].

A better approach, though it comes with its own challenges, is to have the ATP produce a formal proof in a format that can be imported and translated into the ITP's kernel language. This way, the ITP can validate the ATP's result without redoing the entire proof search, preserving the small trusted kernel design. Besides allowing ITPs to query ATPs for proofs of conjectures, such a format also allows formally checking the proofs of theorem provers, for example in competitions, and to build libraries of formal proofs, for testing and training of tools.

This section describes SC-TPTP, a format for exchange of first-order proofs between proof systems. The "SC" part of the acronym stands for *Sequent Calculus*, reflecting the underlying deductive system used. The "TPTP" part refers to the *Thousands of Problems for Theorem Provers* library [88], a widely used collection of logical problems and associated standards for representing formulas and proofs. Concretely, SC-TPTP extends the TPTP standard for writing derivations by providing notations and a set of rules to write formal, low-level proofs that can be machine-checked.

?	$\exists$ (existential quantifier)
!	$\forall$ (universal quantifier)
&	$\wedge$ (logical and)
|	$\vee$ (logical or)
⇒	$\Rightarrow$ (logical implication)
⟺	$\Leftrightarrow$ (logical equivalence)
~	$\neg$ (logical negation)
=	equality
#	ϵ (Hilbert's epsilon operator)

Table 3.3: Common elements of the FOFX syntax from TPTP.

In this section we will describe the SC-TPTP format, and the design choices behind it, then describe how to handle common but tricky transformations such as clausification, and finally present the integration in multiple ITPs (Lisa, Lean, HOL Light) and ATPs (Prover9, Goéland, Egg) and the corresponding Lisa tactics.

Acknowledgement of contributions The SC-TPTP format and toolchain were designed and implemented jointly by Julie Cailler and myself. Julie also implemented SC-TPTP in Goéland, and we jointly implemented SC-TPTP output for Prover9. The implementation of SC-TPTP in HOL Light is a contribution of Sankalp Gambhir, and its implementation in Lean is a contribution of Auguste Poiroux.

3.6.1 The SC-TPTP format

The formalism of SC-TPTP is based on sequent calculus for first-order logic, meaning that *statements* are sequents and the basic *inference steps* are sequent calculus rules. Sequent calculus is well-studied and has a number of useful theoretical and practical properties relevant to automated reasoning. Additionally, derivations in tableaux, natural deduction and resolution can be naturally expressed in sequent calculus, making it a convenient target for representing proofs from different ATPs.

A proof is a TPTP *derivation*. A derivation is a list of annotated statements, each of which is equipped with a name, a role, and, in some cases, an indication of how the formula was deduced. The syntax of formulas is that of FOFX formulas from the TPTP standard [91], with a few additions that will be described later. Table 3.3 summarizes common elements of the FOFX syntax.

Example 3.6.1. Given the valid formula $\exists x.\ \forall y.\ (d(x) \Rightarrow d(y))$, an automated prover might produce the following SC-TPTP proof for the *drinker's paradox* conjecture:

```
fof(phi, let, ?[X]: ![Y]: (d(X) ⇒ d(Y))).
fof(f, plain, [d(X), d(Y)] ⟶ [d(Y0), d(Y)],
        inference(hyp, [status(thm), ], [])).
fof(f2, plain, [d(X)] ⟶ [(d(Y) ⇒ d(Y0)), d(Y)],
        inference(rightImplies, [status(thm)], [f])).
```

```
fof(f3, plain, [d(X)] ⟶ [![Y0] : (d(Y) ⇒ d(Y0)), d(Y)],
        inference(rightForall, [status(thm), 0, 'Y0'], [f2])).
fof(f4, plain, [d(X)] ⟶ [$phi, d(Y)],
        inference(rightExists, [status(thm), 0, $fot(Y)], [f3])).
fof(f5, plain, [] ⟶ [$phi, d(X) ⇒ d(Y)],
        inference(rightImplies, [status(thm), ], [f4])).
fof(f6, plain, [] ⟶ [$phi, ![Y]: (d(X) ⇒ d(Y))],
        inference(rightForall, [status(thm), , 'Y'], [f5])).
fof(f7, plain, [] ⟶ [$phi],
        inference(rightExists, [status(thm), 0, $fot(X)], [f6])).
```

In general, an annotated formula in SC-TPTP follows the pattern

```
fof(name, role, statement, annotation)
```

where:

- The **name** of a step is an identifier that later proof steps refer to.
- SC-TPTP recognizes six **roles**:
  - `axiom` indicates that the formula is an axiom and can be used as a leaf in the derivation.
  - `conjecture` does not have a logical meaning in the proof and indicates what the proof is supposed to prove.
  - `simplify` is an alternative to `conjecture`, and indicates that the objective of the problem is to find a minimal representation of the given expression, and prove the equivalence.
  - `plain` indicates a valid statement, deduced from previous statements or axioms.
  - `assumption` is optionally used if the step does not have any premises but is otherwise equivalent to `plain`.
  - `let`, a new role, is used to introduce shorthands, such as `phi` in the first line of the example proof above. `let` statements should not be referred to as premises of future steps; instead, they introduce a new defined constant symbol, which can be used in first-order formulas. Any occurrence of `$phi` is a shorthand (without any variable renaming) of the formula it stands for.

  Let bindings enable structure sharing for proofs, which often need to repeatedly state the same assumptions. Terms can also be bound to identifiers using annotated terms (not present in normal TPTP syntax), e.g. `fot(t, let, f(X, c))`.
- The **statement** is either a formula, or a sequent. A sequent is a pair of sets of formulas of the form `[`$\phi_1, \ldots$`]` ⟶ `[`$\psi_1, \ldots$`]`. A formula is a shortcut for the sequent with an empty left side and one formula on the right side.

- The **annotation** indicates how the formula was derived and is of the form `inference(stepName, [status(thm), p₁, \ldots, pₙ], [r₁, \ldots, rₙ])`. Following the SZS ontologies [87], `status(thm)` indicates the logical status of the formula. In SC-TPTP, all derived statements are `thm`. The $\mathtt{p}_i$ are parameters used to efficiently check and transform the proof, and depend on the specific proof step. The $\mathtt{r}_i$'s are the premises of the steps. They always correspond to names of previous annotated formulas.

Proof step levels The basic steps of sequent calculus, along with substitution of equal terms or equivalent formulas, and the instantiation of free schematic symbols (Table 3.4, Table 3.5), define the logic of SC-TPTP. These steps are low-level and can be efficiently simulated by any proof system that supports first-order logic. However, outputting strict sequent calculus proofs is a stringent requirement for theorem provers. To facilitate the adoption of the format, SC-TPTP proofs allow various proof steps, which are organized into *levels*.

The first level includes exactly the steps from Table 3.4 and Table 3.5, while subsequent levels are flexible and expected to evolve. The second level contains more advanced proof steps for which an algorithm exists to eliminate them, i.e., processing an SC-TPTP proof and unfolding every occurrence of such a step into level 1 steps. Examples of such steps implemented in the SC-TPTP toolchain are listed in Table 3.6.

The third level consists of steps that lack such an implementation but are easily deduced and verified by most proof assistants. With moderate effort, a level 3 step can be converted into a level 2 step. The fourth level contains arbitrary sound deductive steps that cannot be reliably unfolded and may be difficult to implement in a proof-producing form. Examples of level 3 steps can be seen in Table 3.7.

These levels offer useful stepping stones to make a solver proof-producing, without needing to implement every proof reconstruction detail, such as congruence closure or negation normal form, which can be time-consuming.

Schematic symbols SC-TPTP adds[3] schematic formulas, predicates and functions, as we already discussed in Subsection 3.1.1 in the context of λFOL. Semantically, schematic symbols are halfway between constant symbols and variables: they can be instantiated by terms and formulas but cannot be bound. Schematic symbols are the most practical ways to express infinite sets of axioms (which is necessary, for example, to define arithmetic), as well as to express logical theorems that justify logical transformations, for example, that for any formula ϕ, $\phi \wedge \neg\phi \iff \bot$. This makes them essential for a formal proof system such as SC-TPTP. Additionally, schematic symbols are typically supported by first-order proof assistants, such as in Mizar and Lisa. In Metamath [65], schematic variables are referred to as metavariables, while in Isabelle/FOL, higher-order variables in the metalanguage can play this role. In SC-TPTP, schematic predicates and functions start with capital letters.

[3] to TPTP's syntax for first-order formulas

Rule name	Premises	Rule	Parameters
`leftFalse`	0	$\frac{}{\Gamma, \bot \vdash \Delta}$	`i:Int`: Index of $\bot$ on the left
`rightTrue`	0	$\frac{}{\Gamma \vdash \top, \Delta}$	`i:Int`: Index of $\top$ on the right
`hyp`	0	$\frac{}{\Gamma, A \vdash A, \Delta}$	`i:Int`: Index of A on the left `j:Int`: Index of A on the right
`leftWeaken`	1	$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}$	`i:Int`: Index of A on the left
`rightWeaken`	1	$\frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}$	`i:Int`: Index of A on the right
`cut`	2	$\frac{\Gamma \vdash A, \Delta \qquad \Sigma, A \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}$	`i:Int`: Index of A on the right of the first premise
`leftAnd`	1	$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$	`i:Int`: Index of $A \wedge B$ on the left
`leftOr`	2	$\frac{\Gamma, A \vdash \Delta \qquad \Sigma, B \vdash \Pi}{\Gamma, \Sigma, A \vee B \vdash \Delta, \Pi}$	`i:Int`: Index of $A \vee B$ on the left
`leftImplies`	2	$\frac{\Gamma \vdash A, \Delta \qquad \Sigma, B \vdash \Pi}{\Gamma, \Sigma, A \Rightarrow B \vdash \Delta, \Pi}$	`i:Int`: Index of $A \Rightarrow B$ on the left
`leftIff`	1	$\frac{\Gamma, A \Rightarrow B, B \Rightarrow A \vdash \Delta}{\Gamma, A \Leftrightarrow B \vdash \Delta}$	`i:Int`: Index of $A \Leftrightarrow B$ on the left
`leftNot`	1	$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}$	`i:Int`: Index of $\neg A$ on the left
`leftExists`	1	$\frac{\Gamma, A[x := y] \vdash \Delta}{\Gamma, \exists x.A \vdash \Delta}$	`i:Int`: Index of $\exists x.A$ on the left `y:String`: Variable in place of x in the premise
`leftForall`	1	$\frac{\Gamma, A[x := t] \vdash \Delta}{\Gamma, \forall x.A \vdash \Delta}$	`i:Int`: Index of $\forall x.A$ on the left `t:Term`: Term in place of x in the premise
`rightAnd`	2	$\frac{\Gamma \vdash A, \Delta \qquad \Sigma \vdash B, \Pi}{\Gamma, \Sigma \vdash A \wedge B, \Delta, \Pi}$	`i:Int`: Index of $A \wedge B$ on the right
`rightOr`	1	$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta}$	`i:Int`: Index of $A \vee B$ on the right
`rightImplies`	1	$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta}$	`i:Int`: Index of $A \Rightarrow B$ on the right
`rightIff`	2	$\frac{\Gamma \vdash A \Rightarrow B, \Delta \qquad \Sigma \vdash B \Rightarrow A, \Pi}{\Gamma \vdash A \Leftrightarrow B, \Delta}$	`i:Int`: Index of $A \Leftrightarrow B$ on the right
`rightNot`	1	$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}$	`i:Int`: Index of $\neg A$ on the right
`rightExists`	1	$\frac{\Gamma \vdash A[x := t], \Delta}{\Gamma \vdash \exists x.A, \Delta}$	`i:Int`: Index of $\exists x.A$ on the right `t:Term`: Term in place of x in the premise
`rightForall`	1	$\frac{\Gamma \vdash A[x := y], \Delta}{\Gamma \vdash \forall x.A, \Delta}$	`i:Int`: Index of $\forall x.A$ on the right `y:String`: Variable in place of x in the premise
`rightRefl`	0	$\frac{}{\Gamma \vdash t = t, \Delta}$	`i:Int`: Index of $t = t$ on the right

Table 3.4: Level 1 rules of SC-TPTP, part 1.

Rule name	Premises	Rule	Parameters
`rightSubst`	1	$\frac{\Gamma \vdash P(t), \Delta}{\Gamma, t = u \vdash P(u), \Delta}$	`i:Int`: Index of $t = u$ on the left `backward:Int`: If 1, the substitution is done backward `P(Z):Term`: Shape of the predicate on the right `Z:String`: Variable indicating where the substitution takes place
`leftSubst`	1	$\frac{\Gamma, P(t) \vdash \Delta}{\Gamma, t = u, P(u) \vdash \Delta}$	`i:Int`: Index of $t = u$ on the left `backward:Int`: If 1, the substitution is done backward `P(Z):Term`: Shape of the predicate on the left `Z:String`: Variable indicating where the substitution takes place
`rightSubstIff`	1	$\frac{\Gamma \vdash R(\phi), \Delta}{\Gamma, \phi \Leftrightarrow \psi \vdash R(\psi), \Delta}$	`i:Int`: Index of $\phi \Leftrightarrow \psi$ on the left `backward:Int`: If 1, the substitution is done backward `R(Z):Var`: Shape of the predicate on the right `Z:String`: Variable indicating where in P the substitution takes place
`leftSubstIff`	1	$\frac{\Gamma, R(\phi) \vdash \Delta}{\Gamma, \phi \Leftrightarrow \psi, R(\psi) \vdash \Delta}$	`i:Int`: Index of $\phi \Leftrightarrow \psi$ on the left `backward:Int`: If 1, the substitution is done backward `R(Z):Var`: Shape of the predicate on the right `Z:String`: Schematic formula indicating where in P the substitution takes place
`instFun`	1	$\frac{\Gamma[F_X] \vdash \Delta[F_X]}{\Gamma[F_X := t_X] \vdash \Delta[F_X := t_X]}$	`'F': String`: Schematic function to substitute. `t:Term`: Term, possibly containing $X_1, \ldots, X_n$, to instantiate F with `Xs: Seq[String]`: Variables parametrizing t. The length gives the arity of F
`instPred`	1	$\frac{\Gamma[P_X] \vdash \Delta[P_X]}{\Gamma[P_X := \phi_X] \vdash \Delta[P_X := \phi_X]}$	`'P': String`: Schematic predicate to substitute. ϕ`: Formula`: Formula, possibly containing $X_1, \ldots, X_n$, to instantiate F with `Xs: Seq[String]`: Variables parametrizing ϕ. The length gives the arity of P
`rightEpsilon`	1	$\frac{\Gamma \vdash A[x := t], \Delta}{\Gamma \vdash A[x := \epsilon x.A], \Delta}$	`A:Formula`: Formula defining the epsilon-term `X:String`: Variable being substituted in A `t:Term`: Term in place of x in the premise
`leftEpsilon`	1	$\frac{\Gamma, A[x := y] \vdash \Delta}{\Gamma, A[x := \epsilon x.A] \vdash \Delta}$	`i:Int`: Index of $A[x := y]$ on the left of the premise `y:String`: Variable in place of x in the premise

Table 3.5: Level 1 rules of SC-TPTP, part 2.

Rule name	Premises	Rule	Parameters
`congruence`	0	$\dfrac{}{\Gamma, \Delta}$	No parameters Γ contains a set of ground equalities such that P and Q are congruents
`leftHyp`	0	$\dfrac{}{\Gamma, A, \neg A \vdash \Delta}$	`i:Int`: Index of A on the left
`leftNotAnd`	2	$\dfrac{\Gamma, \neg A \vdash \Delta \qquad \Sigma, \neg B \vdash \Pi}{\Gamma, \Sigma, \neg(A \wedge B) \vdash \Delta, \Pi}$	`i:Int`: Index of $\neg(A \wedge B)$ on the left
`leftNotOr`	1	$\dfrac{\Gamma, \neg A, \neg B \vdash \Delta}{\Gamma, \neg(A \vee B) \vdash \Delta}$	`i:Int`: Index of $\neg(A \vee B)$ on the left
`leftNotImplies`	1	$\dfrac{\Gamma, A, \neg B \vdash \Delta}{\Gamma, \neg(A \Rightarrow B) \vdash \Delta}$	`i:Int`: Index of $\neg(A \Rightarrow B)$ on the left
`leftNotIff`	2	$\dfrac{\Gamma, \neg(A \Rightarrow B) \vdash \Delta \qquad \Sigma, \neg(B \Rightarrow A) \vdash \Pi}{\Gamma, \Sigma, \neg(A \Leftrightarrow B) \vdash \Delta, \Pi}$	`i:Int`: Index of $\neg(A \Leftrightarrow B)$ on the left
`leftNotNot`	1	$\dfrac{\Gamma, A \vdash \Delta}{\Gamma, \neg\neg A \vdash \Delta}$	`i:Int`: Index of $\neg\neg A$ on the left
`leftNotEx`	1	$\dfrac{\Gamma, \neg A[x := t] \vdash \Delta}{\Gamma, \neg\exists x.A \vdash \Delta}$	`i:Int`: Index of $\neg\exists x.A$ on the left `t:Term`: Term in place of x in the premise
`leftNotAll`	1	$\dfrac{\Gamma, \neg A \vdash \Delta}{\Gamma, \neg\forall x.A \vdash \Delta}$	`i:Int`: Index of $\neg\forall x.A$ on the left `y:String`: Variable in place of x in the premise

Table 3.6: Current Level 2 rules of SC-TPTP, which can be unfolded into level 1 rules with the SC-TPTP utils.

Rule name	Premises	Rule	Parameters
`rightReflIff`	0	$\frac{}{\Gamma \vdash A \Leftrightarrow A, \Delta}$	`i:Int`: Index of $A \Leftrightarrow A$ on the right
`rightSubstMulti`	1	$\frac{\Gamma \vdash P(\vec{t}), \Delta}{\Gamma \vdash P(\vec{u}), \Delta}$	`i_`, ..., `i_n`: Index of formulas $t_j = u_j$ on the left `P(Z_, ... , Z_n):Term`: Shape of the formula on the right `Z_`, ..., `Z_n`: Var: variables indicating where to substitute
`leftSubstMulti`	1	$\frac{\Gamma, P(\vec{t}) \vdash \Delta}{\Gamma, P(\vec{u}) \vdash \Delta}$	`i_`, ..., `i_n`: Index of formula $t_j = u_j$ on the left `P(Z_, ... , Z_n):Term`: Shape of the formula on the left `Z_`, ..., `Z_n`: Var: variables indicating where to substitute
`rightSubstEqForall`	2	$\frac{\Gamma \vdash R(\phi(t)), \Delta \quad \Sigma \vdash \forall x.\phi(x) = \psi(x), \Pi}{\Gamma, \Sigma \vdash R(\psi(t)), \Delta, \Pi}$	`i:Int`: Index of $\forall x.\phi(x) = \psi(x)$ on the right of the second premise `R(Z):Var`: Shape of the predicate on the right `Z:Var`: Variable indicating where in P the substitution takes place
`rightSubstIffForall`	2	$\frac{\Gamma \vdash R(\phi(t)), \Delta \quad \Sigma \vdash \forall x.\phi(x) \Leftrightarrow \psi(x), \Pi}{\Gamma, \Sigma \vdash R(\psi(t)), \Delta, \Pi}$	`i:Int`: Index of $\forall x.\phi(x) \Leftrightarrow \psi(x)$ on the right of the second premise `R(Z):Var`: Shape of the predicate on the right `Z:Var`: Variable indicating where in P the substitution takes place
`rightNnf`	1	$\frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \phi', \Delta}$	`i:Int`: Index of ϕ on the right of the premise `j:Int`: Index of ϕ' on the right of the conclusion ϕ and ϕ' have the same negation normal form
`rightPrenex`	1	$\frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \phi', \Delta}$	`i:Int`: Index of ϕ on the right of the premise `j:Int`: Index of ϕ' on the right of the conclusion ϕ and ϕ' have the same prenex normal form
`clausify`	0	$\frac{}{\Gamma, a \iff b \circ c \vdash \Delta}$	`i:Int`: Index of $a \iff b \circ c$ on the left Δ is a clause resulting from the inequality $a \iff b \circ c$
`elimIffRefl`	1	$\frac{\Gamma, \forall x_1, ..., x_n.\phi \iff \phi \vdash \Delta}{\Gamma \vdash \Delta}$	`i:Int`: Index of $\phi \iff \phi$ on the left of the premise
`res`	2	$\frac{\Gamma \vdash A, \Delta \qquad \Sigma \vdash \neg A, \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}$	`i:Int`: Index of A on the right of the first premise
`instMult`	1	$\frac{\Gamma[F_1, ..., F_n], \vdash \Delta[F_1, ..., F_n]}{\Gamma[G_1, ..., G_n] \vdash \Delta[G_1, ..., G_n]}$	Sequence of triplets of the form: `'F': String, t: Term|Formula, Xs: Seq[Str]`. Each triplet has the same construction as arguments of `instFun` and `instPred`, but the substitution is carried simultaneously. Simultaneous substitution of function and predicate schemas, including variables and formula variables.

Table 3.7: Current Level 3 rules of SC-TPTP, which are used in the exporting of proofs from Goéland, Prover 9 and Egg, and easily reconstructed by proof assistants.

`classical`	The proof is carried in classical logic and contains sequents with multiple formulas on the right-hand side, or uses higher level steps only valid in classical logic.
`epsilon`	The proof makes use of ϵ-terms.
`propext`	Usually, substitution of equivalent formulas (`leftSubstIff` and `rightSubstIff`) is a metatheorem of first-order logic and can be unfolded. However, this is not the case if substitution is carried below epsilon quantifiers. In this case, propositional extensionality is required.
`schem`	The proof uses schematic predicates or functions, which start with a capital letter.
`let`	The proof uses defined expressions starting with `$` that are defined with the `let` role
`fot`	The proof uses top-level annotated terms, with the `let` or `simplify` role.

Table 3.8: Annotations defined for fine-grained logical properties. We expect more will exist in the future.

Epsilon terms Another important addition to SC-TPTP is the support for reasoning with Hilbert's epsilon operator ϵ, as already discussed in Subsection 3.1.2. As a reminder, the ϵ operator is a term-level binder with the following introduction rule:

$$\frac{\Gamma \vdash \Delta, \phi}{\Gamma \vdash \Delta, \phi[x := (\epsilon x.\phi)]}\ \texttt{rightEpsilon}$$

In Lisa, our main motivation for description operators was to enrich our syntax with higher-order operators, and to enable definitions. In SC-TPTP, our main motivation is to allow efficient certification of Skolemization. Constructive deskolemization, the problem of computing a proof of a formula ϕ from a proof of a formula ϕ', where ϕ' is the result of applying Skolemization to ϕ, is hard: all known methods have exponential or non-elementary blow-up [3], depending on whether inner or outer Skolemization is used. This happens even when the proof is cut-free [6].

An epsilon-term $\epsilon x.\phi$ can be seen as a Skolem function $f(y_1, \ldots, y_n)$ where y_i are all the free variables of ϕ except for x. This enables certification of Skolemization in purely deductive proofs with a linear number of substitution steps. The downside is that the target system must support epsilon terms, but this is the case in many proof assistants[4]. SC-TPTP proofs use `#` to denote ϵ.

Logic options to classify proofs SC-TPTP uses annotations in the TPTP header to indicate logical features used in the proof and advanced SC-TPTP features. Such logical annotations (Table 3.8) are written similarly to the *Specialist Problem Class* (SPC) annotations in TPTP. A declaration:

[4]It is called `The` in Mizar, `@` in HOL Light and HOL4, ϵ in Lisa, `Some` in Isabelle and `epsilon` in Rocq and Lean (requires non-constructive axioms).

```
% Logic    : classical_epsilon_schem_let
```

for example, indicates that the proof uses classical logic (`classical`), permitting sequents with multiple formulas on the right-hand side. (`epsilon`) indicates use of the epsilon choice operator, (`schem`) indicates use of schematic predicates or functions, and (`let`) enables defined expressions introduced with the `let` role.

3.6.2 Simulating non-deductive proofs

Some common proof strategies used by ATPs are not *deductive*, meaning they do not derive a true conclusion from true premises but instead transform the entire problem. The most common such transformations are *proofs by contradiction*, *Tseitin transformation* and *Skolemization*. Most proof assistants, however, are based on purely deductive logic and cannot accept or prove these steps. When applied correctly, these strategies preserve soundness, and since sequent calculus is complete, any statement proven using non-deductive strategies will also have a proof in strict sequent calculus. However, finding a pure sequent calculus proof can be computationally difficult and time-consuming. Deskolemization, in particular, can blow up the size of the proofs without ϵ [3, 6].

Proof assistants typically have purely deductive systems, and their kernels typically do not accept such steps. But they are necessary to transform a formula into clausal normal form, which many ATPs rely on. These classes of steps cannot simply be defined as logical steps in SC-TPTP and unfolded locally. Eliminating them from an ATP proof requires modifying the proof globally. In this section we explain how proofs by contradiction, Tseitin transformation and Skolemization can be simulated deductively. We have implemented these techniques in the SC-TPTP toolchain to enable representation of Prover9 proofs.

Proof by contradiction with backward and forward proofs

A common strategy to prove a conjecture ϕ is to assume $\neg\phi$ and deduce $\bot$. In TPTP proofs, this is typically denoted as:

```
fof(c, conjecture, φ).
fof(nc, negated_conjecture, ~φ).
...
fof(fn, plain, $false).
```

From such a proof, we want to recover a proof of the actual conjecture ϕ. More generally, suppose a set of axioms $\phi_1, \ldots, \phi_n$ and a conjecture ϕ are given. We may assume $\neg\phi$ and proceed forward:

$$\dfrac{\dfrac{\dfrac{\vdash \psi_1 \quad \ldots \quad \vdash \psi_n \quad \dfrac{}{\neg\phi \vdash \neg\phi}\,\texttt{hyp}}{\vdots}}{\neg\phi \vdash}}{\vdash \neg\neg\phi}\,\texttt{rightNot}}{\vdash \phi}\,\texttt{simp}$$

As a formula and its universal closure are interdeducible, we assume without loss of generality and for simplicity that ϕ has no free variables. $\vdots$ denotes the proof of $\bot$ from the axioms and $\neg\phi$. It can proceed entirely with the $\neg\phi$ formula present on the left, without any effect on any proof step.

Clausification is the transformation of a formula into a conjunction of disjunctions of literals. There are two steps in this transformation: Skolemization and the Tseitin transformation. Neither is a deductive step; they return equisatisfiable formulas rather than logical consequences of the input. Tseitin transformation can theoretically be replaced by a conjunctive normal form using distributivity, but at an exponential cost. For Skolemization, a proof of a Skolemized formula can be non-elementarily smaller than a proof of the non-Skolemized formula. Hence, for practical use it is necessary to be able to represent these non-deductive operations efficiently.

Tseitin transformation on axioms and negated conjecture

Suppose we have a formula ϕ given as an axiom, and we aim to deduce $\bot$. The structure of the desired proof from the clausified axioms will be as follows:

$$\dfrac{\vdash a_1 \vee \ldots \vee a_n \qquad \ldots \qquad \vdash z_1 \vee \ldots \vee z_n}{\dfrac{\vdots}{\vdash \bot}}$$

where $a_1 \vee \ldots \vee a_n$, …, $z_1 \vee \ldots \vee z_n$ are the clauses resulting from the Tseitin transformation of ϕ. But the clauses are not in general consequences of ϕ, as the transformation only preserves satisfiability.

Suppose ϕ contains $a \wedge b$ as a subterm. We can simulate the first step of Tseitin transformation as follows:

$$\dfrac{\dfrac{\vdash \phi(a \wedge b)}{A \Leftrightarrow (a \wedge b) \vdash \phi(A)}\ 1.\ \texttt{rightSubstIff} \qquad \dfrac{}{A \Leftrightarrow (a \wedge b) \vdash \neg A \vee a}\ 2. \qquad \dfrac{}{A \Leftrightarrow (a \wedge b) \vdash \neg A \vee b}\ 3. \qquad \dfrac{}{A \Leftrightarrow (a \wedge b) \vdash A \vee \neg a \vee \neg b}\ 4.}{\dfrac{\dfrac{\dfrac{\vdots}{A \Leftrightarrow (a \wedge b) \vdash \psi}}{(a \wedge b) \Leftrightarrow (a \wedge b) \vdash \psi}\ 5.\ \texttt{instPred}}{\vdash \psi}\ 6.\ \texttt{elimIffRefl}}$$

Steps 2, 3 and 4 are constant-size tautologies. Then the resolution proof follows normally, independently of the $A \iff (a \wedge b)$ on the left-hand side.

Finally, we eliminate the assumption. Step 5 allows instantiating the schematic formula A with an arbitrary formula, and steps 5 and 6 eliminate the now trivial assumption $(a \wedge b) \Leftrightarrow (a \wedge b)$ from the left-hand side. To justify equisatisfiability, we need to ensure that A is fresh. This is enforced here by step 5. If A was not a fresh symbol but already appearing in ψ, for example, the initial part of the proof would still work, but step 5 would fail.

This process can be repeated as many times as needed, until the axioms and negated conjectures are themselves clauses. Note that every single step can be unfolded into a constant number of level 1 steps, making the whole proof of Tseitin transformation linear. All steps are deductive and can be locally checked, independently of the rest of the proof.

Skolemization

Consider a formula in prenex normal form with a single alternation of quantifiers:

$$\phi := \forall x.\exists y.\psi(x, y)$$

where ψ is quantifier-free. Its Skolemized form is the formula

$$\phi^{\mathsf{sko}} := \forall x.\psi(x, f(x))$$

with f a fresh function symbol. Then, $\phi \vdash \bot$ has a proof if and only if ϕ^{sko} has a proof. Similarly, the empty sequent has a proof from ϕ if and only if it has a proof from ϕ^{sko}. However, in general the proof of the original formula is complicated to compute and may be exponentially or even non-elementarily larger than the proof of the Skolemized formula.

We can formalize Skolemization to make it locally checkable and purely deductive by introducing Hilbert's ε choice operator, denoted `#` in SC-TPTP. ε is a term-level binder intuitively selecting an element satisfying a predicate, if such an element exists, and an arbitrary term otherwise. The following is a provable consequence of the definition of the `rightEpsilon` rule:

$$\frac{}{\exists x.P(x) \Leftrightarrow P(\varepsilon x.P(x))}\ \texttt{substEpsilon}$$

When applied to the formula ϕ above, we obtain the following proof.

Example 3.6.2.
Conjecture: $\vdash \forall x.\exists y.\psi(x, y)$
Proof:

$$\dfrac{\dfrac{\dfrac{\mathcal{A}}{\forall x.\psi(x, f(x))}}{\forall x.\psi(x, \varepsilon y.\psi(x, y))}}{\forall x.\exists y.\psi(x, y)}\ \texttt{substEpsilon}$$

Observe that the subproof $\mathcal{A}$ can ignore the nature of ε and treat it as a black box, exactly as if it were a fresh function symbol $f(x)$: an uninterpreted function symbol containing the same free variables as the ε expression. Hence, if $\mathcal{A}$ is a proof of ϕ^{sko}, the above is a proof of ϕ. Examples of SC-TPTP proofs relying on these techniques to prove a formula in clausal form can be found in the supplementary material. Additionally, all the techniques above have been implemented in the SC-TPTP toolchain, which can automatically transform proofs relying on these non-deductive steps into purely deductive SC-TPTP proofs. In particular, this allows Prover9, which is based on resolution and hence requires clausification, to produce SC-TPTP proofs.

3.6.3 Tools connected through SC-TPTP

We have made extensions for multiple proof systems to produce or validate SC-TPTP proofs. The overall architecture is illustrated in Figure 3.11. The theorem provers we

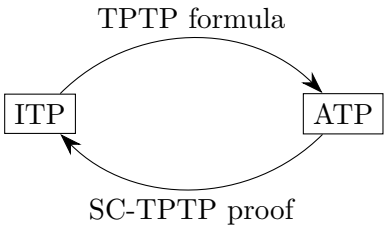


Figure 3.11: SC-TPTP for hammers.

made able to import SC-TPTP proofs are Lisa, HOL Light and Lean. The automated theorem provers we made able to produce SC-TPTP proofs are Prover9, Goéland and egg.

Automated theorem provers producing SC-TPTP proofs

The following ATPs are able to produce proofs in the SC-TPTP format, which can be readily verified by an ITP.

Goéland Goéland [17, 16] is an automated theorem prover for first-order logic with equality. It relies on a concurrent tableaux-based proof-search procedure that allows it to conduct a fair branch exploration. Tableaux proofs are close in spirit to sequent proofs. Section A.3 provides an example of a proof produced by Goéland.

Prover9 Prover9 [61] is an automated theorem prover for first-order logic with equality that implements the resolution and paramodulation calculi, thus relying on clausification. We implement proof production for Prover9 in two steps. First, we assume the input problem is clausal, that is, the conjecture is `$false` and each axiom is a single clause `[a, b, ...]`, naturally represented as the sequent `[]` $\longrightarrow$ `[a, b, ...]`. The steps of a resolution proof are easy to map to proof steps of SC-TPTP: the resolution step is similar to a `cut` step, and the instantiation is exactly the `instFun` step. Then, we separately implement certification of clausification, which modifies the output proof, as described in Subsection 3.6.2. The resulting procedure takes arbitrary first-order formulas as input and outputs verifiable SC-TPTP proofs. This procedure is part of the SC-TPTP utils toolchain and can be reused for other ATPs relying on clausification.

egg egg [99] (short for E-Graphs Good) is not a traditional ATP but a tool implementing high-performance E-graphs [71] and saturation, often used to optimize programs. It can, however, be seen as solving a fragment of first-order logic with equality, where the equivalence relation is the union of $\Leftrightarrow$ and $=$.

egg outputs explanations for pairs of equivalent expressions. We add TPTP input for egg, which supports quantified equalities and equivalences as assumptions and conjectures, and transforms egg justifications into SC-TPTP proofs.

egg is also able to find the smallest representative in an equivalence class, motivating

the use of SC-TPTP for more than yes/no answers. This motivates the experimental `simplify` role for annotated formulas and terms, exclusive to conjectures, which prompts the solver to find a simplified version of the given term or formula, and prove its equivalence to the original. For example:

```
fof(a, axiom, (! [Xx]: (Xx = sf(sf(sf(sf(sf(Xx)))))))).
fof(a2, axiom, (! [Xx]: (Xx = sf(sf(sf(Xx)))))).
fot(c, simplify, (p(sf(c)))).
...
fof(f5, plain, [] ⟶ [(p(sf(c)) ⟺ p(c))], inference( ... )).
```

This feature is currently supported only by egg. Section A.3 provides examples of proofs produced by our egg wrapper, with level 1 and level 2 steps.

Interactive theorem provers validating SC-TPTP proofs

Lisa Lisa implements three tactics, respectively `Goeland`, `Prover9` and `Egg`, which export a sequent to SC-TPTP, invoke the respective ATP, and reconstruct the proof within Lisa. The following are examples of Lisa invocations of Goéland, Prover9 and egg:

```
1  val divide_mult_shift = Theorem((
2          ∀(x, x/t ⩵ x), ∀(x, ∀(y, x/y ⩵ t/(y/x))),
3          ∀(x, ∀(y, (x/y)*y ⩵ x))) ⊢ ((t2/t3)*(t3/t2))/t ⩵ t):
4      have(thesis) by Egg
5
6  val drinkers2 = Theorem(∃(x, ∀(y, d(x) ⟹ d(y)))):
7      have(thesis) by Goeland
8
9  val thm = Theorem((∀(x, P(x)) ∨ ∀(y, Q(y))) ⟹ (P(∅) ∨ Q(∅)) ):
10     have(thesis) by Prover9
```

HOL Light HOL Light [47] is an LCF-style interactive theorem prover based on higher-order logic implemented in OCaml. We contribute an extensible way to generically construct interfaces to external provers for use in HOL Light, and a TPTP parser in OCaml using `ocamllex` and `menhir` supporting the FOFX fragment with the SC-TPTP extensions described here. Our implementation provides a function `sctptp_tac`, which, given a way to construct invocations to an external solver, can be instantiated in one line as a reusable tactic. Instantiations `EGG`, `GOELAND` and `PROVER9` for the respective ATPs are included. Each accepts a list of existing theorems to use as axioms, and a sequent to prove, and returns a `thm` object proving the sequent if a proof was found and reconstructed. The following are examples of HOL Light invocations of Goéland and egg:

```
(* syntax: PROVER [premises] [antecedents] conclusion *)
# let drinkers = GOELAND [] [] `?x:ind. !y. d(x) ⟹ d(y)`;;
val drinkers : thm = ⊢ exists x. forall y. d x ⟹ d y
```

```
# let div_mult_shift = EGG [] [
    `!x. x/t = x`; `!x y. x/y = t/(y/x)`; `!x y. (x/y)*y = x`
    ] `((t2/t3)*(t3/t2))/t = t`;;
val div_mult_shift : thm =
  forall v3 v5. v3/v5 = t/(v5/v3), forall v3 v5. v3/v5 * v5 = v3,
  forall v3. v3/t = v3 ⊢ (t2/ t3 * t3/t2) / t = t
```

To transform the original problem (possibly containing higher-order terms) into a first-order SC-TPTP query, we adapted the monomorphization procedure from the existing implementation of the Meson tactic in HOL Light. All higher-order terms are abstracted into named untyped first-order constants. During proof reconstruction, however, the types for all terms must be recovered. To this end, we implement a type inference and unification procedure similar to that of the simply-typed λ-calculus.

Lean Lean [27] is an interactive theorem prover based on dependent type theory. In our framework, proof reconstruction is implemented using the standard Lean 4 tactic interface. We developed custom tactics, `egg`, `prover9` and `goeland`, that automate the entire workflow, including sequent export to TPTP, invocation of the underlying ATP and subsequent parsing, reification, reconstruction and checking of the proof within Lean 4, allowing, e.g., the following use.

```
example ( α : Type) [Nonempty α] (d : α → Prop) :
  ∃ y : α, ∀ x : α, (d x → d y) := by goeland
```

Our development uses the monomorphization and reification tools from the Lean Auto project [57]. We adapt its mechanism for exporting formulas from the TH0 to the FOF format within the TPTP framework. In the translation of TPTP expressions to Lean expressions, we use type inference to recover information about types of variables. Our proof reconstruction system is implemented using a backward proof strategy within a single global context, allowing us to leverage the Lean 4 tactic proof mode.

SC-TPTP utilities and central repository

To support the SC-TPTP format, we released a library of tools and utilities to handle SC-TPTP proofs at *https://github.com/SC-TPTP/sc-tptp*. The library provides a parser and an independent proof checker for SC-TPTP proofs, as well as functions able to unfold level 2 proof steps into level 1 proofs, such as a proof-producing implementation of an e-graph able to unfold the `congruence` proof step. It also contains a module able to certify clausification and extend proofs of formulas in CNF provided by ATPs relying on clausification to form a complete proof of the original formula, which is central to SC-TPTP support for Prover9. The library further contains helpers, examples, test cases and documentation. The repository also contains links and forks of tools with SC-TPTP support. Section A.3 contains examples of SC-TPTP proofs.

This concludes our presentation of SC-TPTP, a proof format for representing proofs in first-order logic with equality, enabling verification of proofs produced by first-order

ATPs and transfer of first-order proofs between proof systems. We demonstrated how clausification can be represented in the format and implemented SC-TPTP interfaces in several types of automated and interactive theorem provers.

4 Conclusion

In this thesis, we have developed two distinct but related main topics. First, we have developed the approach of *orthologic-based reasoning* for verification pipelines, which consists in using, in key areas, components that are incomplete but smaller, faster and more predictable than general-purpose SAT/SMT solving, while still providing mathematically precise guarantees of completeness. The motivating observation was that propositional logic acts as the glue in formal verification, and that significant friction and slowdowns come from the need for normalization, simplification and transformation of formulas across syntactic variations that may seem obvious to users, and the sometimes unpredictable behaviour that comes from the necessary heuristics of tools designed to solve a fundamentally intractable problem.

The goal was not to replace SAT/SMT solvers, which have long served a crucial role in verification systems, but rather to propose a complementary approach that can be used in parts of verification systems where the full power of SAT/SMT solvers is not needed, and where predictability, efficiency and trust are more valuable.

We have achieved this combination of clear completeness guarantees, asymptotic efficiency and practical applicability, through the algebraic class of ortholattices, and its corresponding logic, *orthologic*. Ortholattices are a strict weakening of Boolean algebras that do not satisfy the distributivity law, and as such satisfy the requirement of being a sound approximation of propositional logic with clear, algebraic completeness guarantees. We have shown that propositional formulas can be normalized with respect to the laws of ortholattices in quadratic time, and that the entailment problem in orthologic can be decided in time $\mathcal{O}(n^2(1 + |A|))$, where n is the size of the formulas and $|A|$ is the size of a set of assumptions. Hence, with a low degree polynomial-time complexity, these algorithms satisfy the efficiency requirements. Finally, we demonstrated the practical usability of orthologic-based reasoning in a variety of verification applications where efficiency and predictability are highly valuable: when used in the kernel of the Lisa proof assistant, we obtain shorter proofs, less user frustration and faster library development; in Stainless, orthologic-based normalization of verification conditions makes formulas shorter before solving and improves cache hit rate, resulting in significantly lower verification times; in type systems with union and intersection types, orthologic entailment provides a simpler, faster and more predictable approach to subtyping than

in popular programming languages.

Furthermore, we have developed a broad theoretical foundation of orthologic, with the hope that it will facilitate new applications and further adoption in verification systems. We have shown that orthologic enjoys the interpolation property, which hints at applications in model checking. We have extended orthologic with monotonic function symbols, which was important as a basis for type systems, where they correspond to covariant and contravariant type constructors. We have studied the proof strength of orthologic with axioms, and shown that it captures, not only soundly but also completely, significant classes of propositional formulas used in verification, such as Horn clauses. We have extended orthologic with predicates and term-level variables, and showed that this extension remains decidable and has exponentially better worst-case complexity than its classical counterpart. These results suggest applications in logic programming.

Our main practical achievement that leverages orthologic-based reasoning is the proof assistant *Lisa*. Its logical kernel integrates orthologic equivalence as a form of definitional equality, saving users and tactics alike from tedious propositional rewrites, and making proofs shorter and more robust. Orthologic is a core part of Lisa, which in turn is the original motivation for orthologic. It may seem that in the end Lisa is a project largely independent of other developments in orthologic, and while it is true that it has developed its own research directions (proof transfer, set-theoretic type systems, the development of λFOL), they are united and motivated by the same core principles.

More generally, this thesis emphasizes recurring engineering constraints that shape verification engineering and proof systems: *trust*, *predictability*, *efficiency* and *usability*. The aim for these qualities guided the design of orthologic-based reasoning. The particular choices of theoretical results and algorithms we pursued were not arbitrary, but rather guided by the aim of satisfying these aspirational goals. These same principles continuously guided the development of Lisa, alongside two additional ones that are more specific to proof assistants: *programmability*, to offer users the full power of modern programming, and *interoperability*, to exchange proofs and theorems across systems. Every particular design choice of Lisa can be seen as an attempt to optimally balance these sometimes conflicting principles.

Five years after the beginning of Lisa's development, and following multiple iterations over its foundations, reimplementations of its kernel, redesigns of its proof language and rewrites of its library, Lisa stands further than I could have hoped. Thanks to the contributions of many people, key results of foundational set theory have been formalized — including non-trivial theorems such as the theory of ordinals with the transfinite recursion theorem as well as the formalization of dependent function spaces. These allowed us to integrate important mathematical features of a type-theoretic nature such as algebraic data types (ADT) with induction principles, tactics for automated type checking and replay of proofs in higher-order logic (specifically HOL Light). On the automation side, I have implemented a tableaux-based tactic for first-order logic that leverages orthologic normalization, a tactic for congruence closure based on proof-producing e-graphs adapted to λFOL, a tactic for propositional reasoning based on orthologic normalization

and more.

With the goal of leveraging powerful external automated theorem provers to help populate Lisa's library with elementary developments, we have designed a new format for the exchange of proofs across proof systems, called *SC-TPTP*. Concerns about lack of interoperability have long been present in the background of the ITP and ATP communities. By implementing support for SC-TPTP in a number of systems with very different foundations and implementations (Lisa based on set theory, Lean on type theory and HOL Light on higher-order logic, for proof assistants; Goéland based on tableaux, Prover9 based on resolution and egg based on e-graphs, for automated theorem provers), we demonstrated the practicality of the approach and established a baseline for future developments in proof interoperability.

Lisa has also been an instrument for teaching classes at EPFL. Students in the graduate course *Formal Verification* used it for the past three years to formalize mathematical theories such as group theory, ring theory, topology and hyperreal numbers. Additionally, Lisa was used for the past three years by the teaching team (of which I was not myself a part) of the undergraduate course *Functional Programming* (about 400 students) as an autograder and interface to teach undergraduate students how to formally reason about functional programs. These renewed successes highlight the potential of Lisa, and, more broadly, of proof assistants, for teaching.

4.1 Future work

This thesis does not mark the end of research in orthologic-based reasoning. We have shown that orthologic with quantifiers does not admit quantifier elimination, but whether the logic is decidable — and if so, at what complexity — remains open. More importantly, even though the quadratic-time bound for the word problem is tight in our algorithm, it has not been proven that no faster algorithm can exist. Interestingly, this question is still open even in the case of lattices, and establishing such a lower bound would be a significant theoretical result.

Most of the open research directions lie in the practical applications of orthologic-based reasoning. We have already mentioned the relevance of the interpolation property for model checking. Another promising direction is an extension of Datalog: the fragment of effectively propositional orthologic restricted to Horn clauses corresponds to the syntax of Datalog, and using the orthologic entailment procedure gives the same answers as a Datalog engine. Then, allowing arbitrary propositional formulas would yield a proper extension of Datalog whose semantics would not be classical, but still purely *logical*, unlike semantics based on negation as failure or fixpoints. For such an application to approach the practical efficiency of Datalog engines, significant engineering work would be needed to specialize the algorithms for orthologic entailment, but the theoretical foundations are already in place.

We have presented a simple typed language whose type system supports intersection, union and negation types and whose subtyping relation is based on provability in ortho-

logic. Existing programming languages that support these features exhibit undesirable behaviour such as incompleteness and inefficiency. I am convinced that this principled approach of relying on provability using entailment in ortholattices (or lattices, if negation is not desired) is practically viable and a reasonably low-hanging fruit. The next theoretical step is to design a type inference algorithm for this system, and the next practical one is a complete implementation of such a programming language to serve as a proof of concept.

Type systems are also the next step for Lisa. In principle, it should be possible to design a *set-theoretic type system* and implement it as a soft type system over set theory where types correspond to sets and functions to set-theoretic functions. We should then be able to implement common type-theoretic constructs in this framework: we have already done it for algebraic data types, and work is ongoing to support dependent types and universes, leading to the calculus of constructions (CoC) with universes. By adding support for inductive types and well-founded recursion, we would obtain a system subsuming the calculus of inductive constructions (CIC) underlying Rocq and Lean. This would be a significant formalization effort, requiring higher and more complex reasoning with the ordinal hierarchy than simply algebraic data types, but it would offer a lot of additional expressivity and eventually allow simulating theorems from Rocq and Lean, as we did with HOL Light. Doing this with pure first-order logic would be infeasible in practice, and it is one of the main reasons for the development of λFOL. What would make the type system *set-theoretic* is the inclusion of features inspired by the embedding of types as sets, and most significantly subtyping, corresponding to the subset relation. This is where orthologic-based subtyping comes into play, as it would provide one possible efficient type checking technique. Some questions remain to be studied, such as the treatment of contravariant type constructors, but the big advantage of this approach compared to using type theory as a foundation is that we would be allowed to implement more advanced and risky features without endangering consistency, as the underlying kernel would still ensure soundness. Ultimately, one might argue that a set-theoretic foundation is more suitable to design a type system than pure type theory is itself.

Finally, we cannot deny that one major bottleneck of Lisa is the populating of its library with elementary results. Realistically, this requires more manpower than Lisa may be able to attract or fund in the near future, and additionally I do not think that reimplementing thousands of theorems already formalized in many other systems one more time is the best use of researchers' time. Hence, short of reimplementing Lean's mathlib from scratch, I see two main directions to address this issue.

The first is AI assistance in writing proofs, a growing focus of research in the proof assistant and AI communities, with promising results. Since it is a topic largely independent of the specifics of Lisa, I will not develop it much here, but two key specificities of Lisa in this context are that (i) very little data is available compared to what large language models need, prompting questions about how to generate such data, but on the other hand (ii) the embedding of Lisa in Scala may indirectly provide a lot of related,

syntactically similar data through publicly available Scala codebases.

The second direction, which we have already hinted at, is interoperability. This can take many different forms: directly importing kernel proofs of theorems from other assistants' libraries (as we did with the mechanization of HOL Light); verifying proofs produced by ATPs (what we did with SC-TPTP); leveraging large language models to transfer proofs at the level of proof scripts rather than at the kernel level, a process similar to *code transfer* between different programming languages — which still faces the obstacle of a lack of Lisa data; or automatically translating theorem statements from other proof assistants but not the proofs, and trusting them. This last approach is much simpler but less trustworthy, as it relies on trusting not only the other proof assistant, but also the translation process. It may be a reasonable trade-off to quickly bootstrap Lisa's library with elementary results, and then progressively replace them with fully verified proofs.

Lisa still has a very long way to go before being competitive with established proof assistants such as Lean, and for now it remains a tool to do research *about* proof assistants rather than a tool that mathematicians and software engineers can use productively. However, I believe that between its foundations (set theory is still the standard foundation of mathematics after all) and its design choices (its interface is entirely compatible with Scala programming features and the libraries of Scala and Java, and the perspective of a set-theoretic type system is compelling), Lisa offers many unique features and is sufficiently different from existing systems that it would not be redundant in the landscape of proof assistants. Whether it is Lisa itself, a "Lisa 2.0" or another project yet to be that would build on the lessons learned from Lisa, I see a future where its unique features allow it to stand among the proof assistants that enable formal verification for mathematicians, computer scientists and software engineers, and perhaps one of the most widely used of them.

A Appendix

A.1 Notation and symbols

Symbol	Description	Definition
Logics and algebraic structures		
OL	Ortholattices	Definition 2.1.15
OL^+	Ortholattices with (anti)monotonic function symbols	Definition 2.1.19
BL^+	Bounded lattices with (anti)monotonic function symbols	Definition 2.1.19
L^+	Lattices with (anti)monotonic function symbols	Definition 2.1.19
BA	Boolean algebras	Definition 2.1.15
$OCBSL$	Orthocomplemented bisemilattices	Definition 2.11.1
QOL	Quantified orthologic	Definition 2.7.1
EPR-OL-D	Effectively propositional orthologic	Definition 2.8.1
CL	Classical logic	—
FOL	First-order logic	—
QBF	Quantified Boolean formulas	—
λFOL	First-order logic with λ-abstraction	Definition 3.1.1
λFOST	First-order set theory with λ-abstraction	Definition 3.2.1
Proof systems		
LK	Gentzen's classical sequent calculus	—
LJ	Gentzen's intuitionistic sequent calculus	—
LO	Sequent calculus for OL	Definition 2.2.2
$\mathbf{LO}^+$	Sequent calculus for OL^+	Definition 2.2.3
$\mathbf{LO}^+_{\boldsymbol{BL}}$	Sequent calculus for BL^+	Definition 2.2.14
$\mathbf{LO}^+_{\boldsymbol{L}}$	Sequent calculus for L^+	Definition 2.2.14
$\mathbf{CF}^+$	Cut-free sequent calculus for OL^+	Definition 2.2.9
$\mathbf{CF}^+_{\boldsymbol{BL}}$	Cut-free sequent calculus for BL^+	Definition 2.2.14

(continued)

Symbol	Description	Definition
$\mathbf{CF}_{\boldsymbol{L}}^{+}$	Cut-free sequent calculus for L^+	Definition 2.2.14
$\mathbf{CF}_{\boldsymbol{\delta}}^{+}$	Cut-free sequent calculus for OL^+ in pseudo negation normal form	Definition 2.4.8
$\mathbf{LO}^{+\boldsymbol{I}}$	Sequent calculus for predicate OL with instantiation (EPR-OL-D)	Definition 2.8.8
QLO	Sequent calculus for QOL	Definition 2.7.1
Terms and formulas		
$\mathcal{T}_K(X)$	Terms with signature K over variables X	Definition 2.1.2
$\|t\|$, $\|T\|$	Size of term t (number of nodes); for a set T, $\sum_{t\in T}\|t\|$	Definition 2.1.3
$F^{i,j,k}$	Function symbol with i invariant, j covariant, k contravariant arguments	Definition 2.1.19
$t[\sigma], T[\sigma]$	Simultaneous substitution of assignment σ into term t (resp. set T)	Definition 2.1.4
$\sigma[x := e]$	Assignment σ updated at x with value e	Definition 2.7.3
$\mathsf{FV}(t)$	Free variables of a term or formula t	Definition 2.1.2
$[\![t]\!]_\alpha$	Interpretation of term t under assignment α	Definition 2.1.6
Algebraic relations		
$s \sim_{\mathcal{A}} t$	s and t have the same interpretation in algebra $\mathcal{A}$	Definition 2.1.6
$s \sim_K t$	s and t have the same interpretation in all algebras of class K	Definition 2.1.7
$s \leq_K t$	s is below t in every lattice of class K	Definition 2.1.17
$[a]_\sim$	Equivalence class of a under $\sim$	Definition 2.1.9
$\mathcal{F}_V(X)$	Free algebra of variety V over variables X	Definition 2.1.11
Sequents and provability		
$\phi^L,\ \phi^R$	Left/right annotated formula in a sequent	Definition 2.2.2
$\vdash_S$	Provability in proof system S	Definition 2.2.2
$\phi \dashv\vdash_S \psi$	ϕ and ψ are mutually derivable in S	Definition 2.2.9
$[\![\phi^L, \psi^R]\!]_\alpha$	Semantics of sequents	Definition 2.2.4
Normalization		
$\mathrm{nnf}(\phi)$	Pseudo negation normal form of ϕ	Subsection 2.4.1
$\overline{K}$	Extension of signature K with dual operators ($\overline{OL^+}$, $\overline{BL^+}$, $\overline{L^+}$)	Subsection 2.4.1
$\beta(\phi)$	Remove complementary pairs from ϕ	Subsection 2.4.2
$\zeta(\phi)$	Remove redundant subformulas from ϕ	Subsection 2.4.2
$\eta(\phi)$	Remove dominated subformulas from ϕ	Subsection 2.4.2

(continued)

Symbol	Description	Definition
Orthologic extensions		
Q_Σ	Atomic formulas over predicate signature Σ	Definition 2.8.1
$d(A)$	Degree of a set of sequents A: maximum number of distinct variables	Definition 2.8.2
s^*, A^*	Expansion of formula/sequent s (resp. set A): all instances by variable substitution	Definition 2.8.2
Term rewriting		
$t \to t'$	Single rewrite step	Definition 2.11.2
$t \xrightarrow{*} t'$	Rewriting in zero or more steps	Definition 2.11.2
$t \xleftrightarrow{*} t'$	Equational closure of rewriting	Definition 2.11.2
λFOL: types and expressions		
$\mathsf{Ind}, \mathsf{Prop}, \tau_1 \to \tau_2$	Types of λFOL	Definition 3.1.1
$\mathrm{ord}(\tau)$	Order of type τ	Definition 3.1.1
$e \sim_\alpha e'$	α-equivalence of expressions	Definition 3.1.1
$e \sim_\beta e'$	β-equivalence of expressions	Definition 3.1.1
$e \sim_\eta e'$	η-equivalence of expressions	Definition 3.1.1
$\lvert e \rvert_\lambda$	Embedding of FOL expression e into λFOL	Definition 3.1.1
HOL embedding into λFOST		
$(\!\lvert e \rvert\!)$	Embedding of HOL expression e into λFOST	Definition 3.5.2
$s \sim_A t$	HOL equality at type A: $(\mathcal{E}\ A)\ s\ t$	Definition 3.5.2
$\mathrm{ctx}^N(t)$	Non-emptiness context of HOL term t	Definition 3.5.2
$\mathrm{ctx}^T(t)$	Typing context of HOL term t	Definition 3.5.2

A.2 List of orthologic proof systems

This appendix collects the proof systems for orthologic used throughout Chapter 2: **LO**, $\mathbf{LO}^+$, $\mathbf{LO}^+_{\boldsymbol{BL}}$, $\mathbf{LO}^+_{\boldsymbol{L}}$, $\mathbf{CF}^+$, $\mathbf{CF}^+_{\boldsymbol{BL}}$, $\mathbf{CF}^+_{\boldsymbol{L}}$, $\mathbf{CF}^+_{\boldsymbol{\delta}}$, $\mathbf{LO}^{+\boldsymbol{I}}$, and **QLO**. Figure A.1 summarizes the relationships between these systems, and the logics they characterize. We list each calculus in full below.

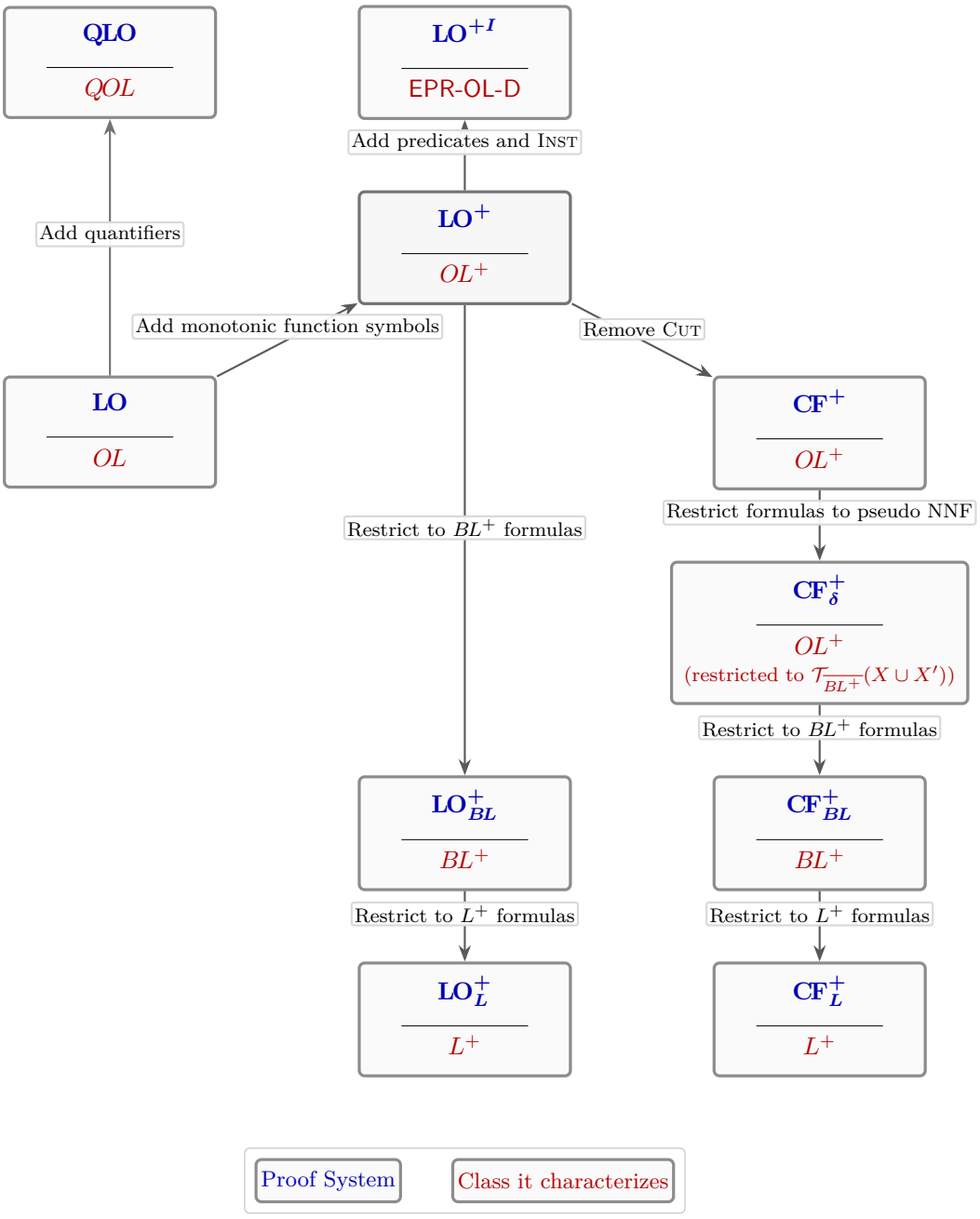


Figure A.1: Overview of proof systems used in Chapter 2.

$$\frac{}{\phi^L, \phi^R}\ \textsc{Hyp}$$

$$\frac{\Gamma, \psi^R \qquad \psi^L, \Delta}{\Gamma, \Delta}\ \textsc{Cut}$$

$$\frac{\Gamma, \Gamma}{\Gamma, \Delta}\ \textsc{Replace}$$

$$\frac{}{\bot^L, \Delta}\ \textsc{LeftBot} \qquad \frac{}{\Gamma, \top^R}\ \textsc{RightTop}$$

$$\frac{\Gamma, \phi^L}{\Gamma, (\phi \wedge \psi)^L}\ \textsc{LeftAnd} \qquad \frac{\Gamma, \phi^R \qquad \Gamma, \psi^R}{\Gamma, (\phi \wedge \psi)^R}\ \textsc{RightAnd}$$

$$\frac{\Gamma, \phi^L \qquad \Gamma, \psi^L}{\Gamma, (\phi \vee \psi)^L}\ \textsc{LeftOr} \qquad \frac{\Gamma, \phi^R}{\Gamma, (\phi \vee \psi)^R}\ \textsc{RightOr}$$

$$\frac{\Gamma, \phi^R}{\Gamma, (\neg\phi)^L}\ \textsc{LeftNot} \qquad \frac{\Gamma, \phi^L}{\Gamma, (\neg\phi)^R}\ \textsc{RightNot}$$

$$\frac{}{\Gamma, \Delta}\ \textsc{Axiom} \quad \text{if } (\Gamma, \Delta) \in A$$

Figure A.2: The proof system **LO** for orthologic (Definition 2.2.2).

$$\frac{}{\phi^L, \phi^R}\ \textsc{Hyp}$$

$$\frac{\Gamma, \psi^R \qquad \psi^L, \Delta}{\Gamma, \Delta}\ \textsc{Cut}$$

$$\frac{\Gamma, \Gamma}{\Gamma, \Delta}\ \textsc{Replace}$$

$$\frac{}{\bot^L, \Delta}\ \textsc{LeftBot} \qquad \frac{}{\Gamma, \top^R}\ \textsc{RightTop}$$

$$\frac{\Gamma, \phi^L}{\Gamma, (\phi \wedge \psi)^L}\ \textsc{LeftAnd} \qquad \frac{\Gamma, \phi^R \qquad \Gamma, \psi^R}{\Gamma, (\phi \wedge \psi)^R}\ \textsc{RightAnd}$$

$$\frac{\Gamma, \phi^L \qquad \Gamma, \psi^L}{\Gamma, (\phi \vee \psi)^L}\ \textsc{LeftOr} \qquad \frac{\Gamma, \phi^R}{\Gamma, (\phi \vee \psi)^R}\ \textsc{RightOr}$$

$$\frac{\Gamma, \phi^R}{\Gamma, (\neg\phi)^L}\ \textsc{LeftNot} \qquad \frac{\Gamma, \phi^L}{\Gamma, (\neg\phi)^R}\ \textsc{RightNot}$$

$$\frac{\phi^L_{x_1}, \psi^R_{x_1} \qquad \psi^L_{x_1}, \phi^R_{x_1} \quad ... \qquad \phi^L_{y_1}, \psi^R_{y_1} \quad ... \qquad \psi^L_{z_1}, \phi^R_{z_1} \quad ...}{F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)^L, F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)^R}\ \text{F-rule}$$

$$\frac{}{\Gamma, \Delta}\ \textsc{Axiom} \quad \text{if } (\Gamma, \Delta) \in A$$

Figure A.3: The proof system $\mathbf{LO}^+$ for orthologic with (anti)monotonic function symbols (Definition 2.2.3).

$$\frac{}{\phi^L, \phi^R}\ \textsc{Hyp}$$

$$\frac{\gamma^L, \psi^R \qquad \psi^L, \delta^R}{\gamma^L, \delta^R}\ \textsc{Cut}$$

$$\frac{}{\bot^L, \gamma^R}\ \textsc{LeftBot} \qquad \frac{}{\gamma^L, \top^R}\ \textsc{RightTop}$$

$$\frac{\phi^L, \gamma^R}{(\phi \wedge \psi)^L, \gamma^R}\ \textsc{LeftAnd} \qquad \frac{\gamma^L, \phi^R \qquad \gamma^L, \psi^R}{\gamma^L, (\phi \wedge \psi)^R}\ \textsc{RightAnd}$$

$$\frac{\phi^L, \gamma^R \qquad \psi^L, \gamma^R}{(\phi \vee \psi)^L, \gamma^R}\ \textsc{LeftOr} \qquad \frac{\gamma^L, \phi^R}{\gamma^L, (\phi \vee \psi)^R}\ \textsc{RightOr}$$

$$\frac{\phi^L_{x_1}, \psi^R_{x_1} \qquad \psi^L_{x_1}, \phi^R_{x_1} \quad ... \qquad \phi^L_{y_1}, \psi^R_{y_1} \quad ... \qquad \psi^L_{z_1}, \phi^R_{z_1} \quad ...}{F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)^L, F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)^R}\ \text{F-rule}$$

$$\frac{}{\gamma^L, \delta^R}\ \textsc{Axiom} \quad \text{with } (\gamma^L, \delta^R) \in A$$

Figure A.4: The proof system $\mathbf{LO}^+_{\boldsymbol{BL}}$ for the logic of bounded lattices with (anti)monotonic function symbols (Definition 2.2.14).

$$\frac{}{\phi^L, \phi^R}\ \textsc{Hyp}$$

$$\frac{\gamma^L, \psi^R \qquad \psi^L, \delta^R}{\gamma^L, \delta^R}\ \textsc{Cut}$$

$$\frac{\phi^L, \gamma^R}{(\phi \wedge \psi)^L, \gamma^R}\ \textsc{LeftAnd} \qquad \frac{\gamma^L, \phi^R \qquad \gamma^L, \psi^R}{\gamma^L, (\phi \wedge \psi)^R}\ \textsc{RightAnd}$$

$$\frac{\phi^L, \gamma^R \qquad \psi^L, \gamma^R}{(\phi \vee \psi)^L, \gamma^R}\ \textsc{LeftOr} \qquad \frac{\gamma^L, \phi^R}{\gamma^L, (\phi \vee \psi)^R}\ \textsc{RightOr}$$

$$\frac{\phi^L_{x_1}, \psi^R_{x_1} \qquad \psi^L_{x_1}, \phi^R_{x_1} \quad ... \quad \phi^L_{y_1}, \psi^R_{y_1} \quad ... \quad \psi^L_{z_1}, \phi^R_{z_1} \quad ...}{F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)^L, F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)^R}\ \textsc{F-rule}$$

$$\frac{}{\gamma^L, \delta^R}\ \textsc{Axiom} \quad \text{with } (\gamma^L, \delta^R) \in A$$

Figure A.5: The proof system $\mathbf{LO}^+_{\boldsymbol{L}}$ for the logic of bounded lattices with (anti)monotonic function symbols (Definition 2.2.14).

$$\frac{}{\phi^L, \phi^R}\ \textsc{Hyp}$$

$$\frac{\Gamma, \Gamma}{\Gamma, \Delta}\ \textsc{Replace}$$

$$\frac{}{\bot^L, \Delta}\ \textsc{LeftBot} \qquad \frac{}{\Gamma, \top^R}\ \textsc{RightTop}$$

$$\frac{\Gamma, \phi^L}{\Gamma, (\phi \wedge \psi)^L}\ \textsc{LeftAnd} \qquad \frac{\Gamma, \phi^R \qquad \Gamma, \psi^R}{\Gamma, (\phi \wedge \psi)^R}\ \textsc{RightAnd}$$

$$\frac{\Gamma, \phi^L \qquad \Gamma, \psi^L}{\Gamma, (\phi \vee \psi)^L}\ \textsc{LeftOr} \qquad \frac{\Gamma, \phi^R}{\Gamma, (\phi \vee \psi)^R}\ \textsc{RightOr}$$

$$\frac{\Gamma, \phi^R}{\Gamma, (\neg\phi)^L}\ \textsc{LeftNot} \qquad \frac{\Gamma, \phi^L}{\Gamma, (\neg\phi)^R}\ \textsc{RightNot}$$

$$\frac{\phi^L_{x_1}, \psi^R_{x_1} \qquad \psi^L_{x_1}, \phi^R_{x_1} \quad ... \quad \phi^L_{y_1}, \psi^R_{y_1} \quad ... \quad \psi^L_{z_1}, \phi^R_{z_1} \quad ...}{F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)^L, F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)^R}\ \textsc{F-rule}$$

$$\frac{\Gamma, \phi^R \qquad \psi^L, \Delta}{\Gamma, \Delta}\ \textsc{AxiomCut} \quad (\text{with } (\phi, \psi) \in A)$$

Figure A.6: The cut-free proof system $\mathbf{CF}^+$ for orthologic with (anti)monotonic function symbols (Definition 2.2.9).

$$\frac{}{\phi^L, \phi^R}\ \textsc{Hyp}$$

$$\frac{\phi^L, \gamma^R}{(\phi \wedge \psi)^L, \gamma^R}\ \textsc{LeftAnd} \qquad \frac{\gamma^L, \phi^R \qquad \gamma^L, \psi^R}{\gamma^L, (\phi \wedge \psi)^R}\ \textsc{RightAnd}$$

$$\frac{\phi^L, \gamma^R \qquad \psi^L, \gamma^R}{(\phi \vee \psi)^L, \gamma^R}\ \textsc{LeftOr} \qquad \frac{\gamma^L, \phi^R}{\gamma^L, (\phi \vee \psi)^R}\ \textsc{RightOr}$$

$$\frac{\phi^L_{x_1}, \psi^R_{x_1} \qquad \psi^L_{x_1}, \phi^R_{x_1} \quad ... \qquad \phi^L_{y_1}, \psi^R_{y_1} \quad ... \qquad \psi^L_{z_1}, \phi^R_{z_1} \quad ...}{F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)^L, F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)^R}\ \text{F-rule}$$

$$\frac{\gamma^L, \phi^R \qquad \psi^L, \delta^R}{\gamma^L, \delta^R}\ \textsc{AxiomCut} \quad (\text{with } (\phi, \psi) \in A)$$

Figure A.7: The cut-free proof system $\mathbf{CF}^+_{\boldsymbol{BL}}$ for the logic of bounded lattices with (anti)monotonic function symbols (Definition 2.2.14).

$$\frac{}{\phi^L, \phi^R}\ \textsc{Hyp}$$

$$\frac{\phi^L, \gamma^R}{(\phi \wedge \psi)^L, \gamma^R}\ \textsc{LeftAnd} \qquad \frac{\gamma^L, \phi^R \qquad \gamma^L, \psi^R}{\gamma^L, (\phi \wedge \psi)^R}\ \textsc{RightAnd}$$

$$\frac{\phi^L, \gamma^R \qquad \psi^L, \gamma^R}{(\phi \vee \psi)^L, \gamma^R}\ \textsc{LeftOr} \qquad \frac{\gamma^L, \phi^R}{\gamma^L, (\phi \vee \psi)^R}\ \textsc{RightOr}$$

$$\frac{\phi^L_{x_1}, \psi^R_{x_1} \qquad \psi^L_{x_1}, \phi^R_{x_1} \quad ... \qquad \phi^L_{y_1}, \psi^R_{y_1} \quad ... \qquad \psi^L_{z_1}, \phi^R_{z_1} \quad ...}{F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)^L, F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)^R}\ \text{F-rule}$$

$$\frac{\gamma^L, \phi^R \qquad \psi^L, \delta^R}{\gamma^L, \delta^R}\ \textsc{AxiomCut} \quad (\text{with } (\phi, \psi) \in A)$$

Figure A.8: The cut-free proof system $\mathbf{CF}^+_{\boldsymbol{L}}$ for the logic of bounded lattices with (anti)monotonic function symbols (Definition 2.2.14).

$$\frac{}{\phi^L, \phi^R}\ \textsc{Hyp}$$

$$\frac{}{\phi^L, (\neg\phi)^L}\ \textsc{LeftHyp} \qquad \frac{}{\phi^R, (\neg\phi)^R}\ \textsc{RightHyp}$$

$$\frac{\Gamma, \Gamma}{\Gamma, \Delta}\ \textsc{Replace}$$

$$\frac{}{\bot^L, \Delta}\ \textsc{LeftBot} \qquad \frac{}{\Gamma, \top^R}\ \textsc{RightTop}$$

$$\frac{\Gamma, \phi^L}{\Gamma, (\phi \wedge \psi)^L}\ \textsc{LeftAnd} \qquad \frac{\Gamma, \phi^R \qquad \Gamma, \psi^R}{\Gamma, (\phi \wedge \psi)^R}\ \textsc{RightAnd}$$

$$\frac{\Gamma, \phi^L \qquad \Gamma, \psi^L}{\Gamma, (\phi \vee \psi)^L}\ \textsc{LeftOr} \qquad \frac{\Gamma, \phi^R}{\Gamma, (\phi \vee \psi)^R}\ \textsc{RightOr}$$

$$\frac{\phi^L_{x_1}, \psi^R_{x_1} \qquad \psi^L_{x_1}, \phi^R_{x_1} \quad ... \qquad \phi^L_{y_1}, \psi^R_{y_1} \quad ... \qquad \psi^L_{z_1}, \phi^R_{z_1} \quad ...}{F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)^L, F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)^R}\ \textsc{F-rule}$$

$$\frac{\Gamma, \phi^R \qquad \psi^L, \Delta}{\Gamma, \Delta}\ \textsc{AxiomCut} \quad \text{(with } (\phi, \psi) \in A\text{)}$$

Figure A.9: The cut-free proof system $\mathbf{CF}^+_{\boldsymbol{\delta}}$ for orthologic with formulas in pseudo negation normal form (Definition 2.4.8).

$$\frac{}{\phi^L, \phi^R}\ \textsc{Hyp}$$

$$\frac{\Gamma, \psi^R \qquad \psi^L, \Delta}{\Gamma, \Delta}\ \textsc{Cut}$$

$$\frac{\Gamma, \Gamma}{\Gamma, \Delta}\ \textsc{Replace}$$

$$\frac{}{\bot^L, \Delta}\ \textsc{LeftBot} \qquad \frac{}{\Gamma, \top^R}\ \textsc{RightTop}$$

$$\frac{\Gamma, \phi^L}{\Gamma, (\phi \wedge \psi)^L}\ \textsc{LeftAnd} \qquad \frac{\Gamma, \phi^R \qquad \Gamma, \psi^R}{\Gamma, (\phi \wedge \psi)^R}\ \textsc{RightAnd}$$

$$\frac{\Gamma, \phi^L \qquad \Gamma, \psi^L}{\Gamma, (\phi \vee \psi)^L}\ \textsc{LeftOr} \qquad \frac{\Gamma, \phi^R}{\Gamma, (\phi \vee \psi)^R}\ \textsc{RightOr}$$

$$\frac{\Gamma, \phi^R}{\Gamma, (\neg\phi)^L}\ \textsc{LeftNot} \qquad \frac{\Gamma, \phi^L}{\Gamma, (\neg\phi)^R}\ \textsc{RightNot}$$

$$\frac{\phi^L_{x_1}, \psi^R_{x_1} \qquad \psi^L_{x_1}, \phi^R_{x_1} \quad ... \qquad \phi^L_{y_1}, \psi^R_{y_1} \quad ... \qquad \psi^L_{z_1}, \phi^R_{z_1} \quad ...}{F(\phi_{x_1}, \ldots, \phi_{y_1}, \ldots, \phi_{z_1}, ...)^L, F(\psi_{x_1}, \ldots, \psi_{y_1}, \ldots, \psi_{z_1}, ...)^R}\ \text{F-rule}$$

$$\frac{}{\Gamma, \Delta}\ \textsc{Axiom} \quad \text{if } (\Gamma, \Delta) \in A$$

$$\frac{\Gamma, \Delta}{\Gamma[\vec{x} := \vec{t}], \Delta[\vec{x} := \vec{t}]}\ \textsc{Inst}$$

Figure A.10: The proof system $\mathbf{LO}^{+\boldsymbol{I}}$ for predicate orthologic with instantiation (Definition 2.8.8).

$$\frac{}{\phi^L, \phi^R}\ \textsc{Hyp}$$

$$\frac{\Gamma, \psi^R \qquad \psi^L, \Delta}{\Gamma, \Delta}\ \textsc{Cut}$$

$$\frac{\Gamma, \Gamma}{\Gamma, \Delta}\ \textsc{Replace}$$

$$\frac{}{\bot^L, \Delta}\ \textsc{LeftBot} \qquad \frac{}{\Gamma, \top^R}\ \textsc{RightTop}$$

$$\frac{\Gamma, \phi^L}{\Gamma, (\phi \wedge \psi)^L}\ \textsc{LeftAnd} \qquad \frac{\Gamma, \phi^R \qquad \Gamma, \psi^R}{\Gamma, (\phi \wedge \psi)^R}\ \textsc{RightAnd}$$

$$\frac{\Gamma, \phi^L \qquad \Gamma, \psi^L}{\Gamma, (\phi \vee \psi)^L}\ \textsc{LeftOr} \qquad \frac{\Gamma, \phi^R}{\Gamma, (\phi \vee \psi)^R}\ \textsc{RightOr}$$

$$\frac{\Gamma, \phi^R}{\Gamma, (\neg\phi)^L}\ \textsc{LeftNot} \qquad \frac{\Gamma, \phi^L}{\Gamma, (\neg\phi)^R}\ \textsc{RightNot}$$

$$\frac{\Gamma, \phi[x := \psi]^L}{\Gamma, (\bigwedge x.\ \phi)^L}\ \textsc{LeftForall} \qquad \frac{\Gamma, \phi[x := x']^R}{\Gamma, (\bigwedge x.\ \phi)^R}\ \textsc{RightForall}\ (x' \text{ not free in } \Gamma)$$

$$\frac{\Gamma, \phi[x := x']^L}{\Gamma, (\bigvee x.\ \phi)^L}\ \textsc{LeftExists}\ (x' \text{ not free in } \Gamma) \qquad \frac{\Gamma, \phi[x := \psi]^R}{\Gamma, (\bigvee x.\ \phi)^R}\ \textsc{RightExists}$$

Figure A.11: The proof system **QLO** for quantified orthologic (Definition 2.7.1).

A.3 Examples of SC-TPTP proofs

Listing A.1: Example proof from egg: Level 2.

```
%----------------------------------------------------------------------
% Status   : Theorem
% SPC      : FOF_UNK_RFO_SEQ
% Solver   : egg v0.9.5
%          : egg-sc-tptp v0..0
% Logic    : schem
%----------------------------------------------------------------------
fof(div_one, axiom, ! [X]: d(X, t) = X).
fof(cancel_den, axiom, ! [X, Y]: (m(d(X, Y), Y) = X)).
fof(invert_div, axiom, ! [X, Y]: d(X, Y) = d(t, d(Y, X))).
fof(c, conjecture, d(m(d(t2, t3), d(t3, t2)), t) = t).


fof(f0, plain, [] --> [
    d(m(d(t2, t3), d(t3, t2)), t) = d(m(d(t2, t3), d(t3, t2)), t)],
    inference(rightRefl, [...], [])).
fof(f, plain, [] --> [
    d(m(d(t2, t3), d(t3, t2)), t) = m(d(t2, t3), d(t3, t2))],
    inference(rightSubstEqForall, [...], [div_one, f0])).
fof(f2, plain, [] --> [
    d(m(d(t2, t3), d(t3, t2)), t) = m(d(t, d(t3, t2)), d(t3, t2))],
    inference(rightSubstEqForall, [...], [invert_div, f])).
fof(f3, plain, [] --> [
    d(m(d(t2, t3), d(t3, t2)), t) = t],
    inference(rightSubstEqForall, [...], [cancel_den, f2])).
```

Listing A.2: egg proof from the same problem: Level 1.

```
fof(div_one, axiom, ! [X]: d(X, t) = X).
fof(cancel_den, axiom, ! [X, Y]: (m(d(X, Y), Y) = X)).
fof(invert_div, axiom, ! [X, Y]: d(X, Y) = d(t, d(Y, X))).
fof(c, conjecture, d(m(d(t2, t3), d(t3, t2)), t) = t).

fof(f0, plain, [] --> [
    d(m(d(t2, t3), d(t3, t2)), t) = d(m(d(t2, t3), d(t3, t2)), t)],
    inference(rightRefl, [...], [])).
fof(f, plain, [
    d(m(d(t2, t3), d(t3, t2)), t) = m(d(t2, t3), d(t3, t2))] --> [
    d(m(d(t2, t3), d(t3, t2)), t) = m(d(t2, t3), d(t3, t2))],
    inference(rightSubst, [...], [f0])).
fof(f2, plain, [![X] : d(X, t) = X] -->
    [d(m(d(t2, t3), d(t3, t2)), t) = m(d(t2, t3), d(t3, t2))],
    inference(leftForall, [...], [f])).
fof(f3, plain, [] -->
    [d(m(d(t2, t3), d(t3, t2)), t) = m(d(t2, t3), d(t3, t2))],
    inference(cut, [...], [div_one, f2])).
fof(f4, plain, [d(t2, t3) = d(t, d(t3, t2))] -->
    [d(m(d(t2, t3), d(t3, t2)), t) = m(d(t, d(t3, t2)), d(t3, t2))],
    inference(rightSubst, [...], [f3])).
fof(f5, plain, [![Y] : d(t2, Y) = d(t, d(Y, t2))] -->
    [d(m(d(t2, t3), d(t3, t2)), t) = m(d(t, d(t3, t2)), d(t3, t2))],
    inference(leftForall, [...], [f4])).
fof(f6, plain, [![X, Y] : d(X, Y) = d(t, d(Y, X))] -->
    [d(m(d(t2, t3), d(t3, t2)), t) = m(d(t, d(t3, t2)), d(t3, t2))],
    inference(leftForall, [...], [f5])).
fof(f7, plain, [] -->
    [d(m(d(t2, t3), d(t3, t2)), t) = m(d(t, d(t3, t2)), d(t3, t2))],
    inference(cut, [...], [invert_div, f6])).
fof(f8, plain, [m(d(t, d(t3, t2)), d(t3, t2)) = t] -->
    [d(m(d(t2, t3), d(t3, t2)), t) = t],
    inference(rightSubst, [...], [f7])).
fof(f9, plain, [![Y] : m(d(t, Y), Y) = t] -->
    [d(m(d(t2, t3), d(t3, t2)), t) = t],
    inference(leftForall, [...], [f8])).
fof(f0, plain, [![X, Y] : m(d(X, Y), Y) = X] -->
    [d(m(d(t2, t3), d(t3, t2)), t) = t],
    inference(leftForall, [...], [f9])).
fof(f, plain, [] -->
    [d(m(d(t2, t3), d(t3, t2)), t) = t],
    inference(cut, [...], [cancel_den, f0])).
```

Listing A.3: Example proof from Goéland.

```
% SZS output start Proof for lisa.maths.Tests.drinkers.p

fof(drinkers, conjecture, (? [X4] : ((! [Y6] : ((d(X4) ⇒ d(Y6)))))).
fof(f9, plain, [~((? [X4] : ((! [Y6] : ((d(X4) ⇒ d(Y6)))))), ~((! [Y6] :
    ((d(X4_8) ⇒ d(Y6)))))), ~((d(X4_8) ⇒ d(Sko_0))), d(X4_8), ~(d(Sko_0)),
     ~((! [Y6] : ((d(Sko_0) ⇒ d(Y6)))))), ~((d(Sko_0) ⇒ d(Sko_))), d(Sko_0
    ), ~(d(Sko_))] ⟶ [],
    inference(leftHyp, [status(thm), 4], [])).
fof(f8, plain, [~((? [X4] : ((! [Y6] : ((d(X4) ⇒ d(Y6)))))), ~((! [Y6] :
    ((d(X4_8) ⇒ d(Y6)))))), ~((d(X4_8) ⇒ d(Sko_0))), d(X4_8), ~(d(Sko_0)),
     ~((! [Y6] : ((d(Sko_0) ⇒ d(Y6)))))), ~((d(Sko_0) ⇒ d(Sko_)))] ⟶ [],
    inference(leftNotImplies, [status(thm), 6], [f9])).
fof(f7, plain, [~((? [X4] : ((! [Y6] : ((d(X4) ⇒ d(Y6)))))), ~((! [Y6] :
    ((d(X4_8) ⇒ d(Y6)))))), ~((d(X4_8) ⇒ d(Sko_0))), d(X4_8), ~(d(Sko_0)),
     ~((! [Y6] : ((d(Sko_0) ⇒ d(Y6)))))] ⟶ [],
    inference(leftNotAll, [status(thm), 5, 'Sko_'], [f8])).
fof(f6, plain, [~((? [X4] : ((! [Y6] : ((d(X4) ⇒ d(Y6)))))), ~((! [Y6] :
    ((d(X4_8) ⇒ d(Y6)))))), ~((d(X4_8) ⇒ d(Sko_0))), d(X4_8), ~(d(Sko_0))]
     ⟶ [],
    inference(leftNotEx, [status(thm), 0, $fot(Sko_0)], [f7])).
fof(f5, plain, [~((? [X4] : ((! [Y6] : ((d(X4) ⇒ d(Y6)))))), ~((! [Y6] :
    ((d(X4_8) ⇒ d(Y6)))))), ~((d(X4_8) ⇒ d(Sko_0)))] ⟶ [],
    inference(leftNotImplies, [status(thm), 2], [f6])).
fof(f4, plain, [~((? [X4] : ((! [Y6] : ((d(X4) ⇒ d(Y6)))))), ~((! [Y6] :
    ((d(X4_8) ⇒ d(Y6)))))] ⟶ [],
    inference(leftNotAll, [status(thm), , 'Sko_0'], [f5])).
fof(f3, plain, [~((? [X4] : ((! [Y6] : ((d(X4) ⇒ d(Y6)))))))] ⟶ [],
    inference(leftNotEx, [status(thm), 0, $fot(X4_8)], [f4]))
fof(f2, plain, [(? [X4] : ((! [Y6] : ((d(X4) ⇒ d(Y6)))))))] ⟶ [(? [X4] :
    ((! [Y6] : ((d(X4) ⇒ d(Y6)))))))],
    inference(hyp, [status(thm), 0], [])).
fof(f, plain, [] ⟶ [(? [X4] : ((! [Y6] : ((d(X4) ⇒ d(Y6)))))), ~((? [X4]
     : ((! [Y6] : ((d(X4) ⇒ d(Y6)))))))],
    inference(rightNot, [status(thm), ], [f2])).
fof(f0, plain, [] ⟶ [(? [X4] : ((! [Y6] : ((d(X4) ⇒ d(Y6)))))))],
    inference(cut, [status(thm), ], [f, f3])).

% SZS output end Proof for lisa.maths.Tests.drinkers.p
```

Bibliography

[1] Oskar Abrahamsson and Magnus O. Myreen. "Fast, Verified Computation for Candle". In: *14th International Conference on Interactive Theorem Proving (ITP 2023)*. Ed. by Adam Naumowicz and René Thiemann. Vol. 268. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 4:1–4:17. ISBN: 978-3-95977-284-6. *doi: 10.4230/LIPIcs.ITP.2023.4*.

[2] Paolo Aglianò. "An Algebraic Investigation of Linear Logic". In: *Archive for Mathematical Logic* 64.5 (July 1, 2025), pp. 893–915. ISSN: 1432-0665. *doi: 10.1007/s00153-025-00969-2*.

[3] J. Avigad. "Eliminating Definitions and Skolem Functions in First-Order Logic". In: *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. Proceedings 16th Annual IEEE Symposium on Logic in Computer Science. June 2001, pp. 139–146. *doi: 10.1109/LICS.2001.932490*.

[4] Jeremy Avigad. *Foundations of Mathematics Mailing List - Harrison Advocates ZFC*. E-mail. May 31, 2018. *https://cs.nyu.edu/pipermail/fom/2018-May/021026.html*.

[5] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge: Cambridge University Press, 1998. ISBN: 978-0-521-77920-3. *doi: 10.1017/CBO9781139172752*.

[6] Matthias Baaz, Stefan Hetzl, and Daniel Weller. "On the Complexity of Proof Deskolemization". In: *The Journal of Symbolic Logic* 77.2 (June 2012), pp. 669–686. ISSN: 0022-4812, 1943-5886. *doi: 10.2178/jsl/1333566645*.

[7] J. L. Bell. "Orthologic, Forcing, and The Manifestation of Attributes". In: *Studies in Logic and the Foundations of Mathematics*. Southeast Asian Conference on Logic. Ed. by C. -T. Chong and M. J. Wicks. Vol. 111. Studies in Logic and the Foundations of Mathematics. Singapore: Elsevier, Jan. 1, 1983, pp. 13–36. *doi: 10.1016/S0049-237X(08)70953-4*.

[8] Jasmin Blanchette et al. "A User's Guide to Sledgehammer for Isabelle/HOL". In: ().

[9] Jasmin C. Blanchette et al. “Hammering towards QED”. In: *Journal of Formalized Reasoning* 9.1 (1 Jan. 29, 2016), pp. 101–148. ISSN: 1972-5787. *doi: 10.6092/issn.1972-5787/4593*.

[10] Olivier Eric Paul Blanvillain. “Abstractions for Type-Level Programming”. Lausanne: EPFL, 2022. 132 pp. *doi: 10.5075/epfl-thesis-8260*.

[11] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Springer Science & Business Media, Aug. 28, 2001. 500 pp. ISBN: 978-3-540-42324-9. Google Books: *3po2Tv_UVcMC*.

[12] Chad Brown. “The Egal Manual”. In: 2014. *https://www.semanticscholar.org/paper/The-Egal-Manual-Brown/05393cbd6e0df18aa8671c05ee4183fff48d6edf*.

[13] Chad E. Brown, C. Kaliszyk, and Karol Pak. “Higher-Order Tarski Grothendieck as a Foundation for Formal Proof”. In: *ITP*. 2019. *doi: 10.4230/LIPIcs.ITP.2019.9*.

[14] Günter Bruns. “Free Ortholattices”. In: *Canadian Journal of Mathematics* 28.5 (Oct. 1976), pp. 977–985. ISSN: 0008-414X, 1496-4279. *doi: 10.4153/CJM-1976-095-6*.

[15] Mario Bucev and Viktor Kunčak. “Formally Verified Quite OK Image Format”. In: *2022 Formal Methods in Computer-Aided Design (FMCAD)*. 2022 Formal Methods in Computer-Aided Design (FMCAD). Oct. 2022, pp. 343–348. *doi: 10.34727/2022/isbn.978-3-85448-053-2_41*.

[16] Julie Cailler. “Designing an Automated Concurrent Tableau-Based Theorem Prover for First-Order Logic”. PhD thesis. Université de Montpellier, Dec. 13, 2023. *https://theses.hal.science/tel-04526940*.

[17] Julie Cailler et al. “Goéland: A Concurrent Tableau-Based Theorem Prover (System Description)”. In: *Automated Reasoning*. Ed. by Jasmin Blanchette, Laura Kovács, and Dirk Pattinson. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 359–368. ISBN: 978-3-031-10769-6. *doi: 10.1007/978-3-031-10769-6_22*.

[18] Giuseppe Castagna. “Programming with Union, Intersection, and Negation Types”. In: *The French School of Programming*. Ed. by Bertrand Meyer. Cham: Springer International Publishing, 2024, pp. 309–378. ISBN: 978-3-031-34518-0. *doi: 10.1007/978-3-031-34518-0_12*.

[19] CDuce. *The CDuce Compiler*. 2021. *https://www.cduce.org*.

[20] Ondřej Čepek and Petr Kučera. “Known and New Classes of Generalized Horn Formulae with Polynomial Recognition and SAT Testing”. In: *Discrete Applied Mathematics*. Boolean and Pseudo-Boolean Functions 149.1 (Aug. 1, 2005), pp. 14–52. ISSN: 0166-218X. *doi: 10.1016/j.dam.2003.12.011*.

[21] Stephen Cook and Tsuyoshi Morioka. “Quantified Propositional Calculus and a Second-Order Theory for NC1”. In: *Archive for Mathematical Logic* 44.6 (Aug. 2005), pp. 711–749. ISSN: 0933-5846, 1432-0665. *doi: 10.1007/s00153-005-0282-2*.

[22] Denis Cousineau et al. “TLA+ Proofs”. In: *FM 2012: Formal Methods 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*. Ed. by Dimitra Giannakopoulou and Dominique Méry. Vol. 7436. FM 2012: Formal Methods 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Paris, France: Springer, Aug. 2012, pp. 147–154. *doi: 10.1007/978-3-642-32759-9_14*.

[23] William Craig. “Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem”. In: *The Journal of Symbolic Logic* 22.3 (Sept. 1957), pp. 250–268. ISSN: 0022-4812, 1943-5886. *doi: 10.2307/2963593*.

[24] Evgeny Dantsin et al. “Complexity and Expressive Power of Logic Programming”. In: *ACM Computing Surveys* 33.3 (Sept. 1, 2001), pp. 374–425. ISSN: 0360-0300. *doi: 10.1145/502807.502810*.

[25] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. 2nd ed. Cambridge: Cambridge University Press, 2002. ISBN: 978-0-521-78451-1. *doi: 10.1017/CBO9780511809088*.

[26] Mark Day et al. “Subtypes vs. Where Clauses: Constraining Parametric Polymorphism”. In: *SIGPLAN Not.* 30.10 (Oct. 17, 1995), pp. 156–168. ISSN: 0362-1340. *doi: 10.1145/217839.217852*.

[27] Leonardo de Moura et al. “The Lean Theorem Prover (System Description)”. In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Cham: Springer International Publishing, 2015, pp. 378–388. ISBN: 978-3-319-21400-9 978-3-319-21401-6. *doi: 10.1007/978-3-319-21401-6_26*.

[28] *Documentation | Flow*. *https://flow.org/en/docs/*.

[29] William F. Dowling and Jean H. Gallier. “Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae”. In: *The Journal of Logic Programming* 1.3 (Oct. 1, 1984), pp. 267–284. ISSN: 0743-1066. *doi: 10.1016/0743-1066(84)90014-1*.

[30] Andrej Dudenhefner and Jakob Rehof. “A Simpler Undecidability Proof for System F Inhabitation”. In: TYPES. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2019, 11 pages. *doi: 10.4230/LIPICS.TYPES.2018.2*.

[31] Andrew Ferraiuolo et al. “Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. New York, NY, USA: Association for Computing Machinery, Oct. 14, 2017, pp. 287–305. ISBN: 978-1-4503-5085-3. *doi: 10.1145/3132747.3132782*.

[32] Ralph Freese, Jaroslav Jezek, and J. Nation. *Free Lattices*. Vol. 42. Mathematical Surveys and Monographs. Providence, Rhode Island: American Mathematical Society, Mar. 6, 1995. ISBN: 978-0-8218-0389-9 978-1-4704-1273-9. *doi: 10.1090/surv/042*.

[33] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. USA: Cambridge University Press, Mar. 1989. 176 pp. ISBN: 978-0-521-37181-0.

[34] R. L. Goodstein. "Mathematical Logic And Hilbert's ε-Symbol". In: *Bulletin of the London Mathematical Society* 2.3 (1970), pp. 366–366. ISSN: 1469-2120. *doi: 10.1112/blms/2.3.366a*.

[35] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Red. by G. Goos et al. Vol. 78. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1979. ISBN: 978-3-540-09724-2 978-3-540-38526-4. *doi: 10.1007/3-540-09724-4*.

[36] Simon Guilloud, Sankalp Gambhir, and Viktor Kunčak. "LISA - A Modern Proof System". In: *14th International Conference on Interactive Theorem Proving (ITP 2023)*. Ed. by Adam Naumowicz and René Thiemann. Vol. 268. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 17:1–17:19. ISBN: 978-3-95977-284-6. *doi: 10.4230/LIPIcs.ITP.2023.17*.

[37] Simon Guilloud, Sankalp Gambhir, and Viktor Kunčak. "Interpolation and Quantifiers in Ortholattices". In: *Verification, Model Checking, and Abstract Interpretation: 25th International Conference, VMCAI 2024, London, United Kingdom, January 15–16, 2024, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, Jan. 15, 2024, pp. 235–257. ISBN: 978-3-031-50523-2. *doi: 10.1007/978-3-031-50524-9_11*.

[38] Simon Guilloud and Viktor Kuncak. *Orthologic with Axioms (Errata)*. Jan. 1, 2024. *https://infoscience.epfl.ch/handle/20.500.14299/201610.2*. Pre-published.

[39] Simon Guilloud and Viktor Kunčak. "Equivalence Checking for Orthocomplemented Bisemilattices in Log-Linear Time". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 196–214. ISBN: 978-3-030-99527-0. *doi: 10.1007/978-3-030-99527-0_11*.

[40] Simon Guilloud and Viktor Kunčak. "Orthologic with Axioms". In: *Proceedings of the ACM on Programming Languages* 8 (POPL Jan. 5, 2024), 39:1150–39:1178. *doi: 10.1145/3632881*.

[41] Simon Guilloud and Clément Pit-Claudel. "Verified and Optimized Implementation of Orthologic Proof Search". In: *Computer Aided Verification*. Ed. by Ruzica Piskac and Zvonimir Rakamarić. Cham: Springer Nature Switzerland, 2025, pp. 130–152. ISBN: 978-3-031-98682-6. *doi: 10.1007/978-3-031-98682-6_8*.

[42] Simon Guilloud et al. “Formula Normalizations in Verification”. In: *Computer Aided Verification*. Ed. by Constantin Enea and Akash Lal. Cham: Springer Nature Switzerland, 2023, pp. 398–422. ISBN: 978-3-031-37709-9. *doi: 10.1007/978-3-031-37709-9_19*.

[43] Simon Guilloud et al. “Mechanized HOL Reasoning in Set Theory”. In: *15th International Conference on Interactive Theorem Proving (ITP 2024)*. Ed. by Yves Bertot, Temur Kutsia, and Michael Norrish. Vol. 309. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 18:1–18:18. ISBN: 978-3-95977-337-9. *doi: 10.4230/LIPIcs.ITP.2024.18*.

[44] Simon Guilloud et al. *Interoperability of Proof Systems with SC-TPTP (Full Version)*. EPFL, 2025. *https://infoscience.epfl.ch/entities/publication/4c3d2db3-3f96-4d83-9794-af6c04673ea8*.

[45] Jad Hamza, Nicolas Voirol, and Viktor Kunčak. “System FR: Formalized Foundations for the Stainless Verifier”. In: *Proceedings of the ACM on Programming Languages* 3 (OOPSLA Oct. 10, 2019), 166:1–166:30. *doi: 10.1145/3360592*.

[46] Jad Hamza et al. “From Verified Scala to STIX File System Embedded Code Using Stainless”. In: *NASA Formal Methods*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Cham: Springer International Publishing, 2022, pp. 393–410. ISBN: 978-3-031-06773-0. *doi: 10.1007/978-3-031-06773-0_21*.

[47] John Harrison. “HOL Light: An Overview”. In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Vol. 5674. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 60–66. ISBN: 978-3-642-03358-2 978-3-642-03359-9. *doi: 10.1007/978-3-642-03359-9_4*.

[48] John Harrison. “Let’s Make Set Theory Great Again!” In: *Axiomatic Set Theory*. Conference on Artificial Intelligence and Theorem Proving. Aussois, 2018, p. 46.

[49] Chris Hawblitzel et al. “Ironclad Apps: End-to-End Security via Automated Full-System Verification”. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 2014, pp. 165–181. ISBN: 978-1-931971-16-4. *https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel*.

[50] Thomas A. Henzinger et al. “Abstractions from Proofs”. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. ACM, 2004, pp. 232–244. *doi: 10.1145/964001.964021*.

[51] Hossein Hojjat and Philipp Rummer. “The ELDARICA Horn Solver”. In: *2018 Formal Methods in Computer Aided Design (FMCAD)* (Oct. 2018), pp. 1–7. *doi: 10.23919/FMCAD.2018.8603013*.

[52] John Hopcroft, Jeffrey UIlman, and Alfred Aho. “The Design And Analysis Of Computer Algorithms”. In: (), p. 479.

[53] *Instability or Regression Bugs: Surprising Behaviors from the Solver/Verifier - Shengyu Huang. https://kumom.io/articles/instability.*

[54] A. B. Kahn. "Topological Sorting of Large Networks". In: *Communications of the ACM* 5.11 (Nov. 1, 1962), pp. 558–562. ISSN: 0001-0782. *doi: 10.1145/368996.369025.*

[55] Gudrun Kalmbach. *Orthomodular Lattices*. London ; New York: Academic Press Inc, Mar. 1, 1983. 390 pp. ISBN: 978-0-12-394580-8.

[56] Daniel Kroening and Georg Weissenbacher. "Interpolation-Based Software Verification with Wolverine". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 573–578. *doi: 10.1007/978-3-642-22110-1_45.*

[57] Lean Prover Community. *Leanprover-Community/Lean-Auto*. leanprover-community, Feb. 16, 2025. *https://github.com/leanprover-community/lean-auto.*

[58] Lionel Parreaux. *Hkust-Taco/Mlscript*. HKUST TACO Lab, June 30, 2025. *https://github.com/hkust-taco/mlscript.*

[59] H. M. MacNeille. "Partially Ordered Sets". In: *Transactions of the American Mathematical Society* 42.3 (1937), pp. 416–460. ISSN: 0002-9947, 1088-6850. *doi: 10.1090/S0002-9947-1937-1501929-X.*

[60] Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. "Contract-Based Resource Verification for Higher-Order Functions with Memoization". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '17. New York, NY, USA: Association for Computing Machinery, Jan. 1, 2017, pp. 330–343. ISBN: 978-1-4503-4660-3. *doi: 10.1145/3009837.3009874.*

[61] William McCune. *Prover9 and Mace4. http://www.cs.unm.edu/~mccune/prover9/%7C.*

[62] K. McMillan and A. Rybalchenko. *Solving Constrained Horn Clauses Using Interpolation*. Microsoft Research, 2013. *https://www.semanticscholar.org/paper/Solving-Constrained-Horn-Clauses-using-McMillan-Micrsoft/3ca20abeccc5d9cb5f1a8043e86aa26560ba056c.*

[63] Kenneth L. McMillan. "Interpolation and SAT-Based Model Checking". In: *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*. Ed. by Warren A. Hunt Jr and Fabio Somenzi. Vol. 2725. Lecture Notes in Computer Science. Springer, 2003, pp. 1–13. *doi: 10.1007/978-3-540-45069-6_1.*

[64] Kenneth L. McMillan. “Interpolants and Symbolic Model Checking”. In: *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*. Ed. by Byron Cook and Andreas Podelski. Vol. 4349. Lecture Notes in Computer Science. Springer, 2007, pp. 89–90. *doi: 10.1007/978-3-540-69738-1_6*.

[65] Norman Megill and David A. Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. Morrisville: Lulu.com, 2019. 248 pp. ISBN: 978-0-359-70223-7.

[66] Andrea Meinander. “A Solution of the Uniform Word Problem for Ortholattices”. In: *Mathematical Structures in Computer Science* 20.4 (Aug. 2010), pp. 625–638. ISSN: 1469-8072, 0960-1295. *doi: 10.1017/S0960129510000125*.

[67] Michel Minoux. “LTUR: A Simplified Linear-Time Unit Resolution Algorithm for Horn Formulae and Computer Implementation”. In: *Information Processing Letters* 29.1 (Sept. 15, 1988), pp. 1–12. ISSN: 0020-0190. *doi: 10.1016/0020-0190(88)90124-X*.

[68] Yutaka Miyazaki. “The Super-Amalgamation Property of the Variety of Ortholattices”. In: *Reports Math. Log.* 33 (1999), pp. 45–63. *https://rml.tcs.uj.edu.pl/rml-33/a-miy-33.htm*.

[69] Adam Naumowicz and Artur Kornilowicz. “A Brief Overview of Mizar”. In: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*. 22nd International Conference on Theorem Proving in Higher Order Logics. Vol. 5674. Aug. 20, 2009, pp. 67–72. ISBN: 978-3-642-03358-2. *doi: 10.1007/978-3-642-03359-9_5*.

[70] Juan A. Navarro and Andrei Voronkov. “Generation of Hard Non-Clausal Random Satisfiability Problems”. In: *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*. AAAI’05. Pittsburgh, Pennsylvania: AAAI Press, July 9, 2005, pp. 436–442. ISBN: 978-1-57735-236-5.

[71] Greg Nelson and Derek C. Oppen. “Fast Decision Procedures Based on Congruence Closure”. In: *Journal of the ACM* 27.2 (Apr. 1, 1980), pp. 356–364. ISSN: 0004-5411. *doi: 10.1145/322186.322198*.

[72] Oded Padon et al. “Paxos Made EPR: Decidable Reasoning about Distributed Protocols”. In: *Proc. ACM Program. Lang.* 1 (OOPSLA Oct. 12, 2017), 108:1–108:31. *doi: 10.1145/3140568*.

[73] Lionel Parreaux and Chun Yin Chau. “MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types”. In: *Proceedings of the ACM on Programming Languages* 6 (OOPSLA2 Oct. 31, 2022), 141:449–141:478. *doi: 10.1145/3563304*.

[74] Lawrence Paulson. *Foundations of Mathematics Mailing List - Harrison Advocates ZFC*. E-mail. June 1, 2018. *https://cs.nyu.edu/pipermail/fom/2018-June/021032.html*.

[75] Lawrence C. Paulson, ed. *Isabelle*. Vol. 828. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer-Verlag, 1994. ISBN: 978-3-540-58244-1. *doi: 10.1007/BFb0030541*.

[76] Gerald E. Peterson and Mark E. Stickel. "Complete Sets of Reductions for Some Equational Theories". In: *J. ACM* 28.2 (Apr. 1, 1981), pp. 233–264. ISSN: 0004-5411. *doi: 10.1145/322248.322251*.

[77] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. "Deciding Effectively Propositional Logic Using DPLL and Substitution Sets". In: *Journal of Automated Reasoning* 44.4 (Apr. 1, 2010), pp. 401–424. ISSN: 1573-0670. *doi: 10.1007/s10817-009-9161-6*.

[78] Aleksandar Prokopec and Martin Odersky. "Conc-Trees for Functional and Parallel Programming". In: *Languages and Compilers for Parallel Computing*. Ed. by Xipeng Shen, Frank Mueller, and James Tuck. Cham: Springer International Publishing, 2016, pp. 254–268. ISBN: 978-3-319-29778-1. *doi: 10.1007/978-3-319-29778-1_16*.

[79] Pavel Pudlák. "The Lengths of Proofs". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 137. Elsevier, 1998, pp. 547–637. ISBN: 978-0-444-89840-1. *doi: 10.1016/S0049-237X(98)80023-2*.

[80] F. P. Ramsey. "On a Problem of Formal Logic". In: *Proceedings of the London Mathematical Society* s2-30.1 (1930), pp. 264–286. ISSN: 1460-244X. *doi: 10.1112/plms/s2-30.1.264*.

[81] John Alan Robinson and Andrei Voronkov, eds. *Handbook of Automated Reasoning (in 2 Volumes)*. MIT: Elsevier and MIT Press, 2001. ISBN: 978-0-444-50813-3. *https://www.sciencedirect.com/book/9780444508133/handbook-of-automated-reasoning*.

[82] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. "Disjunctive Interpolants for Horn-Clause Verification". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer, 2013, pp. 347–363. ISBN: 978-3-642-39799-8. *doi: 10.1007/978-3-642-39799-8_24*.

[83] Joseph R. Shoenfield. *Mathematical Logic*. Natick, Mass: Routledge, 2001. 352 pp. ISBN: 978-1-56881-135-2.

[84] Konrad Slind and Michael Norrish. "A Brief Overview of HOL4". In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer, 2008, pp. 28–32. ISBN: 978-3-540-71067-7. *doi: 10.1007/978-3-540-71067-7_6*.

[85] Matthieu Sozeau. "A New Look at Generalized Rewriting in Type Theory". In: *Journal of Formalized Reasoning* 2.1 (1 2009), pp. 41–62. ISSN: 1972-5787. *doi: 10.6092/issn.1972-5787/1574*.

[86] Stanley Burris. *A Course in Universal Algebra*. 1981. *http://archive.org/details/stanley-burris.-a-course-in-universal-algebra*.

[87] Geoff Sutcliffe. "The SZS Ontologies for Automated Reasoning Software". In: *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008*. Ed. by Piotr Rudnicki et al. Vol. 418. CEUR Workshop Proceedings. CEUR-WS.org, 2008. *https://ceur-ws.org/Vol-418/paper3.pdf*.

[88] Geoff Sutcliffe. "Stepping Stones in the TPTP World". In: *Automated Reasoning*. Ed. by Christoph Benzmüller, Marijn J.H. Heule, and Renate A. Schmidt. Cham: Springer Nature Switzerland, 2024, pp. 30–50. ISBN: 978-3-031-63498-7. *doi: 10.1007/978-3-031-63498-7_3*.

[89] *The Polynomial Freiman-Ruzsa Conjecture*. The Polynomial Freiman-Ruzsa Conjecture. *https://teorth.github.io/pfr//*.

[90] Thomas Jech. *Set Theory*. Springer Monographs in Mathematics. Berlin, Heidelberg: Springer, 2003. ISBN: 978-3-540-44085-7. *doi: 10.1007/3-540-44761-X*.

[91] *TPTP Syntax*. *https://tptp.org/UserDocs/TPTPLanguage/SyntaxBNF.html*.

[92] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. 2nd ed. Cambridge Tracts in Theoretical Computer Science. Cambridge: Cambridge University Press, 2000. ISBN: 978-0-521-77911-1. *doi: 10.1017/CBO9781139168717*.

[93] G. S. Tseitin. "On the Complexity of Derivation in Propositional Calculus". In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Symbolic Computation. Berlin, Heidelberg: Springer, 1983, pp. 466–483. ISBN: 978-3-642-81955-1. *doi: 10.1007/978-3-642-81955-1_28*.

[94] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Vol. 14. Principles of Computer Science Series. New Yor, United States: Computer Science Press, 1988. ISBN: 978-0-7167-8069-4. *https://www.worldcat.org/oclc/310956623*.

[95] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. New Yor, United States: Computer Science Press, 1989. ISBN: 978-0-7167-8162-2.

[96] Pawel Urzyczyn. "Inhabitation in Typed Lambda-Calculi (a Syntactic Approach)". In: *Typed Lambda Calculi and Applications*. Ed. by Philippe Groote and J. Roger Hindley. Red. by Gerhard Goos, Juris Hartmanis, and Jan Leeuwen. Vol. 1210. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 373–389. ISBN: 978-3-540-62688-6 978-3-540-68438-1. *doi: 10.1007/3-540-62688-3_47*.

[97] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. "The Isabelle Framework". In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 33–38. ISBN: 978-3-540-71067-7. *doi: 10.1007/978-3-540-71067-7_7*.

[98] Philip M. Whitman. "Free Lattices". In: *Annals of Mathematics* 42.1 (1941), pp. 325–330. ISSN: 0003-486X. *doi: 10.2307/1969001*. JSTOR: *1969001*.

[99] Max Willsey et al. "Egg: Fast and Extensible Equality Saturation". In: *Proceedings of the ACM on Programming Languages* 5 (POPL Jan. 4, 2021), 23:1–23:29. *doi: 10.1145/3434304*.

[100] V. N. Zemlyachenko, N. M. Korneenko, and R. I. Tyshkevich. "Graph Isomorphism Problem". In: *Journal of Soviet Mathematics* 29.4 (May 1, 1985), pp. 1426–1481. ISSN: 1573-8795. *doi: 10.1007/BF02104746*.