THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE L'UNIVERSITÉ DE MONTPELLIER

En Informatique

École doctorale Information Structure et Système (ED166)

Unité de recherche LIRMM (Laboratoire d'Informatique, de Robotique et de Microéléctronique de Montpellier - UMR5506)

Designing an Automated Concurrent Tableau-Based Theorem Prover for First-Order Logic

Présentée par Julie CAILLER
le 13 décembre 2023

Sous la direction de David DELAHAYE

Devant le jury composé de

Hinde Lilia BOUZIANE Maître de conférences, Université de Montpellier	**Co-encadrante de thèse**
Simon ROBILLARD Maître de conférences, Université de Montpellier	**Co-encadrant de thèse**
Gilles DOWEK Directeur de recherche, Inria	**Rapporteur**
Philipp RÜMMER Professor, University of Regensburg	**Rapporteur**
Marie-Laure MUGNIER Professeure des universités, Université de Montpellier	**Présidente du jury**
Serenella CERRITO Professeure des universités, Université Paris-Saclay	**Examinatrice**
Damien DOLIGEZ Chargé de recherche, Inria	**Examinateur**
Olivier HERMANT Professeur, Mines Paris, Université Paris Sciences et Lettres	**Invité**

UNIVERSITÉ DE MONTPELLIER

Abstract

This thesis describes the design and implementation of a concurrent tableau-based theorem prover called Goéland. Its main feature is the concurrent processing of branches, where each branch searches for its own solution and closes as soon as possible, relying on the agreement mechanism at the parent node.

The main idea behind this prover is to use concurrency to ensure the fairness of a proof-search algorithm in a tableau-based theorem prover. Indeed, multiple choices happen at each step of a tableau proof search, such as the choice of extending a branch, computing a formula, or applying a substitution. While some of them are easy to handle, others are more difficult and require advanced techniques such as backtracking, branch management, and iterative deepening to ensure fair proof search, which makes it challenging to achieve fairness in these conditions. Thanks to its concurrent nature, Goéland overcomes many known fairness issues by providing an elegant approach to manage them. This thesis describes the concurrent proof-search procedure and its key mechanisms for ensuring fairness, including free variable management and eager closure.

Beyond fairness, the completeness of such a procedure is also a crucial element in automated reasoning. In spite of the presence of various provers based on tableaux, only a few of them have demonstrated the completeness of their proof-search procedures, particularly in the context of first-order logic. This thesis provides a completeness proof of the tableau-based calculus implemented in Goéland.

In addition to this solid basis, the prover is also equipped to perform theory reasoning, which plays an active role in current automated reasoning research. Thus, two background reasoners have been developed: an equality reasoner and a module for deduction modulo theory. The former handles reasoning related to equality, while the latter enables Goéland to reason efficiently in any axiomatized theory by utilizing rewrite rules. These rules act as a pre-processor, capturing only the relevant axioms and reducing the search space, resulting in shorter proofs and significant time savings.

In order to expand the capabilities of Goéland, several enhancements have been made. Firstly, polymorphism has been introduced to enable Goéland to reason about typed statements and allow for quantification over types. This improvement proves particularly valuable in certain fields, such as arithmetic reasoning. Furthermore, a proof translation feature has been implemented, enabling Goéland to produce proofs checkable by Coq and Lambdapi.

This thesis presents the design and implementation of Goéland and its various features. It demonstrates the effectiveness of the concurrent proof-search procedure in addressing fairness issues, ensuring its completeness, handling theories through

background reasoners, and incorporating enhancements such as polymorphism and a proof-checkable output. These advancements contribute to the overall efficiency and reliability of Goéland as an interesting tool in the realm of automated reasoning.

Acknowledgements

First of all, I would like to thank my supervisors, David Delahaye, Hinde Lilia Bouziane, and Simon Robillard, for entrusting me with a PhD position and for their support throughout the last three years. Words are probably not enough to express the gratitude and admiration I feel towards you, but I feel very lucky to have the chance to be guided by you through the puzzling yet entertaining world of research. Surprising as it may seem, you even managed to convince me (perhaps involuntarily) to stay in this world.

I extend my thanks to Gilles Dowek and Philipp Rümmer for graciously reviewing my thesis. Thank you very much for the time you spent on my manuscript, for your feedback, and for the subsequent discussions. I also want to thank Marie-Laure Mugnier, Serenella Cerrito, Damien Doligez, and Olivier Hermant for agreeing to be part of my jury. For some of you, additional thanks for the work accomplished or yet to be done in connection with this thesis!

This PhD wouldn't have been one of the best moments of my life without an inspiring environment. I thank all my colleagues at LIRMM, especially the MaREL team, as well as other — far less cool and more obscure — teams like ALGCo or BORÉAL. A special thank to fellow (and sometimes former) PhD students of those teams (I'm not mentioning the permanent staff; doctoral students should have certain privileges too): Giannos, Gaétan, Amadeus, Laure, Bachar, Selena, Guy, Vincent, Nicolas, Pascal, and many others.

Among my colleagues, a special thanks to Maël for his preliminary work on this thesis and for being the only person outside my supervision team to grasp my endeavors during the first year of my PhD. You've got a unique place, and I am so glad it was you. Surprisingly, some students, interns, and engineers ended up making this list grow, so I would like to thank Cédric, Margaux, Matthieu, Dylan, Isaac, and likely many others I may have inadvertently omitted for their work on Goéland. I am also grateful to the students I have had the privilege of teaching, especially Charlotte, Benoît, Robin, Corentin, Josh and Noah. The moments shared with you inspire me to continue in this way, and I hope to consider some of you as colleagues in the future.

Fortunately, some of my longstanding friends have accompanied me throughout all the way. I want to thank Ami, Thibault, Zeta, Fiel, Flo, Maëlle, Xavier, and others I already regret having forgotten (or omitted to keep these acknowledgments from rivaling the thesis length). I let you guess the order in which I put those names :). Thank you for your support, for keeping me in touch with reality, for all the cooking (and eating) times we had, and for being there after all those years. I also

thank my family, who may not necessarily understand everything I have done but keep supporting me anyway!

On top of everything else, a special thank you to Thomas, for being there every day, listening to my complaints, feeding me, and being the worst enemy of my working process. Thank you for being an endless source of motivation and inspiration and for carrying me around with you in everything you do. If those years are the best of my life, it is undoubtedly thanks to you.

Finally, thank you Johann, for your unwavering presence, commitment, support, for our thought-provoking discussions, and so much more. If I could only keep one thing from this journey, it would definitively be you.

Contents

List of Figures

List of Tables

Introduction

With the increasing prevalence and complexity of computer systems, their reliability has become a crucial concern, particularly for critical systems. Any bug or malfunction in these systems can have severe consequences, both in financial and, more importantly, human terms. Some notable examples of bugs include the European Space Agency's Ariane 5 rocket failure and Intel's Pentium FDIV bug.

While *testing* is a standard method to detect and mitigate bugs in a system, it is not exhaustive and cannot guarantee the absence of all defects. On the other hand, *formal methods* provide a means to rigorously demonstrate, through mathematical reasoning, that a system operates precisely as intended without any deviation. Despite being more expensive and time-consuming, formal methods remain the only way to ensure the correctness of a system within its entire operating space.

This assessment hinges on the existence of a *proof* for a given problem. A proof is a sequence of deductive steps aimed at unfolding the reasoning mechanism, leading to the validation of the initial property. Historically, proofs were carried out by a human using traditional pen-and-paper methods. Nevertheless, over time, new techniques have emerged, and now proofs can be established in collaboration with computers, featuring various levels of automation. These degrees of automation span from *proof assistants*, which guide human users in constructing a proof while ensuring error-free derivations, to *theorem provers* that independently and algorithmically generate proofs.

Interactive reasoning tools act like assistants, directing human users in the construction of proofs while guaranteeing the accuracy of the generated derivations. Interactive reasoning plays a significant role in mathematics, assisting mathematicians in proving theorems and verifying essential components in systems with critical implications for human safety. For instance, the `seL4` operating system was verified using the `Isabelle/HOL` proof assistant, and the proof assistant `HOL Light` was a key tool in settling the long-standing Kepler conjecture in 2017.

Automated reasoning tools are able to reason fully or partly automatically about logical formulas. Such reasoning tools also called *theorem provers* (Figure 1), are used extensively in areas including program verification and testing, scheduling, and also to solve problems in mathematics. As an illustration, the *B-Method* was employed to verify the functionality of Paris Metro Line 14. Interactive and automated reasoning

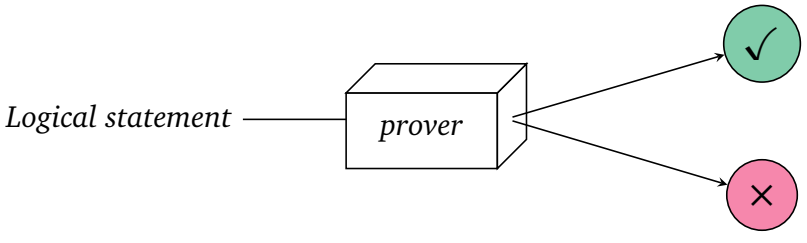


Figure 1: A *prover* takes a logical statement and outputs an information about its truth value.

tools can be used in combination, for example, by outsourcing simpler parts of a proof that is constructed interactively to automated theorem provers.

Although proofs enable us to achieve a satisfactory degree of confidence in our systems, the complexity of the tools for generating them has historically confined their usage to a narrow group of specialists. However, contemporary developments in automated deduction, increased computational capabilities, and efforts to enhance the accessibility of theorem-proving software have popularized these techniques. While it is now easier than ever to collaborate with computers to construct proofs, there is a need to (i) improve usability for end-users and (ii) develop automated tools capable of producing certified proofs. These developments will allow a broader application of formal methods in program verification, contributing to the creation of safer software systems.

This thesis focuses on the design and development of an automated theorem prover. Provers rely on two main characteristics: what they reason about and how they do it. The first aspect relates to the choice of language employed to describe problems, whereas the second refers to the reasoning techniques.

Indeed, automated theorem provers cannot directly engage with software or mathematical problems and first require a translation of these real-world problems into a common computer-readable language, which allows various tools to communicate. *Logic* is such a language. Owing to the wide range and diversity of logics, a myriad of concepts can be represented, spanning from mathematical concepts to real-world scenarios. These logics can be distinguished based on their level of expressiveness and the efficiency of the related reasoning methods.

Certain logics, like propositional logic, are decidable and endowed with effective techniques. However, they may be constrained by a limited degree of expressiveness. In contrast, higher-order logics offer greater expressiveness but often pose challenges for automated reasoning tools. Ultimately, in the context of automation, first-order logic strikes a favorable balance, enabling the representation of individuals and propositions about them. Although semi-decidable, it can be reasoned about efficiently, facilitating deductions from both real-world and mathematical problems, which are translated into logical statements, also called *formulas*.

Language selection significantly influences the choice of the reasoning techniques employed. Different techniques are not universally applicable to all logics, depending on factors such as whether they transform the initial formula. These reasoning techniques are called *proof-search procedures*. As the name suggests, they aim to search for a proof by applying a set of rules on possibly modified versions of the initial formulas, exploring the proof-search space to find a proof.

There exists a multitude of techniques for automated reasoning, each with its unique characteristics. Some of these techniques are known for their efficiency, whereas others possess some properties that can be advantageous in a certain context, such as a specific input or suitability for a broader spectrum of logics.

One of those techniques is the *method of analytic tableaux*. This method operates syntactically by deconstructing the initial formula into subformulas until reaching

axioms known to be true. Notably, this method works with the initial formula, without transformation, making it suitable for interactions with proof assistant and usable in other types of logics. In the context of first-order logic, its primary strength lies in its ability to output a proof, which can be easily translatable to a machine-checkable one.

Challenges

Fairness in Tableau-Based Proof Search

The method of analytic tableaux faces some challenges, which can prevent it from finding a proof. The tree structure generated by breaking down the original formulas into subgoals may introduce dependencies between branches, increasing the complexity of the proof search. Moreover, the variety of choices available at each proof step makes it prone to fairness issues, as stated by Hähnle: "At the present time, no strongly complete, destructive tableau proof procedure is known that works well in practice" [208].

Moreover, most textbooks describe *eager closure* as the standard way to manage closure in free-variable tableaux. In addition to the fairness issue that can be induced by this rule, proving the completeness of a proof-search procedure with eager closure remains difficult since it involves backtracking and, thus, non-monotonicity. However, completeness stands as a critical challenge for every proof-search procedure, as a complete tool instills confidence in its results. In the context of standard completeness proofs for first-order tableau-based procedures, they often involve considering the (infinite) proof tree that would result if the procedure failed to terminate on an unsatisfiable formula, ultimately leading to a counter-model and refutation. However, constructing this infinite derivation is not straightforward for proof-search procedures with backtracking. It is thus hard to achieve a fair, complete, and efficient proof-search procedure in tableaux.

Theory Reasoning in First-Order Logic

Beyond the proof-search procedure itself, certain problems are inherently challenging or demand tailored approaches for specific contexts. This case is particularly present in industrial applications, which often involve explicit constraints or data structure, such as arrays or heaps, or in mathematical theorems pertaining to specific theories, such as set theory. Those problems involve a wide set of axioms, which provide the context necessary to prove the formula, as well as specific semantics or dedicated reasoning techniques, such as those for equality or arithmetic reasoning. With the increase in complexity of the systems, the ability to deal with theories is a crucial concern for any contemporary automated theorem prover.

In the context of first-order logic, theory reasoning is difficult but nonetheless essential. While there are effective methods to tackle specific domains, there is no one-size-fits-all approach to address all possible theories. Moreover, including axioms of theories in problem hypotheses is rarely practical in real-world scenarios, as it

often implies a thoughtless use of axioms that overload the proof-search process. However, strategies have emerged to tackle these challenges. Specifically, although primarily focused on specific theories, *deduction modulo theory* has evolved and can be used as an optimization for automated theorem provers. Transforming axioms into rewrite rules allows to trigger only the relevant ones, leading to a smaller search space and thus to a more efficient proof search. Nonetheless, its integration into a tableau-based proof search is not straightforward, as it closely interacts with critical mechanisms such as free-variable dependencies.

Proof Certification

In a way, automated theorem provers can be seen as oracles, generating an answer for a given formula. While some of them attempt to provide a trace, they can also yield a binary yes/no answer in the worst-case scenario. The trustworthiness of the answer now depends solely on the confidence level we have in the respective ATP. Nevertheless, these tools are typically complex, extensive software in constant evolution, comprising thousands of lines of code and employing sophisticated heuristics. Moreover, since they are developed by humans, they are susceptible to bugs and are inherently error-prone. In such tools, bugs can be disastrous, causing them to prove non-theorems and, consequently, compromising the reliability of the answers they produce. Fortunately, there are two ways to avoid inconsistencies in automated theorem provers: by fully certifying the kernel of the prover using a proof assistant, which is a time-consuming, arduous, and long-term work [212], or by producing machine-checkable proofs, which is generally easily accessible.

The latter relies on the notion of proof certificates. These are proofs generated by an automated theorem prover that an external proof checker can verify. Indeed, in contrast with ATP, proof assistants rely on a certified kernel, ensuring the correctness of proofs checked. Therefore, it is natural to seek a way to combine the strengths of both worlds by producing checkable proofs, thereby instilling full confidence in the results of the ATP. Moreover, relying on an external proof checker to validate proofs significantly enhances the trust we place in them and establishes a common framework for expressing proofs. One advantage of this shared language is the ability to exchange proofs from various theorem provers that may employ different proof systems. However, not all first-order reasoning methods can be readily translated into checkable proofs, especially when advanced heuristics are used.

Contributions

This thesis aims to address a wide range of challenges with the ambition of advancing the field of automated deduction using first-order tableaux. The main contributions encompass both theoretical and practical aspects, the latter leading to developments that implement our theoretical results into a new tool called Goéland. This tool comprises the following key components:

- A tableaux-based concurrent proof-search procedure that ensures fairness by construction. Since the tree structure fits well with concurrent processing, we designed a procedure that explores branches in parallel, addressing fairness issues by leveraging information from one branch to eliminate certain subspaces more quickly. This procedure was proven complete, which is, as far as we know, the first completeness proof of a procedure based on the method of analytic tableaux in first-order logic with eager closure.

- An implementation of two background reasoners to handle theories: an equality reasoner and a deduction modulo theory module. In this thesis, we delve into the incorporation of both a dedicated reasoning module and a more general one, examining their respective interactions with a concurrent tableau-based proof-search procedure.

- A translation procedure from tableaux proofs to a generic machine-checkable proof structure, namely, GS3, as well as two outputs towards dedicated proof assistants: Coq and Lambdapi.

Our main contributions, in addition to the pure implementation work, are manifolds and consequently detailed in separate chapters. Thus, the manuscript is organized as follows. Chapter 1 introduces preliminary notions of logic and concurrency, whereas the related work is available in Chapter 2. The main procedure is offered in Chapter 3 and proved complete in Chapter 4. In order to extend the range of possibilities of Goéland, Chapter 5 presents the theory reasoning embedded into the prover, and Chapter 6 its implementation, as well as that of some other features. Finally, Chapter 7 introduces a strategy to output checkable proofs, which is tested, together with all the functionalities of Goéland, over the TPTP library in Chapter 8.

Chapter 1
Preliminary Notions

Contents

Logic serves as a means to represent the world, with various types available, each differing in expressiveness and efficiency. For instance, *propositional logic* is decidable, enabling quick computation of answers but with limited expressiveness. On the other hand, *first-order logic* is semidecidable, allowing the representation of almost any reasonable notion, although certain problems may remain unprovable. *Higher-order logic* possesses greater expressiveness power but presents challenges in reasoning efficiently, as some key operations in automated reasoning are undecidable in these logics [139].

The choice of logic also influences the reasoning method employed. Usually, automated reasoning is dominated by two main techniques: the *resolution*-based ones and the *tableau*-based ones. Both start with an initial formula and apply reasoning rules to deduce information about it. Resolution-based methods modify the initial formula to gain efficiency, whereas tableau-based ones carry a tree structure and produce a proof of the original problem at the end of its reasoning mechanism.

The characteristic structure of tableau-based methods fits naturally with a concurrent approach. Thus, we are interested in a method allowing us to work on all the branches simultaneously. In order to do this, this chapter introduces some notions of concurrent algorithmic and then delves into a mechanism to ensure communication between branches. A specific concurrent semantic is also presented and used to describe the procedure in Chapter 3.

This chapter presents the basic material for this thesis. It begins by defining

first-order logic in Section 1.1 and then focuses in Section 1.2 on a particular proof method, the *method of analytic tableaux*. Lastly, Section 1.3 introduces elements of concurrent algorithmic.

1.1 First-Order Logic

Thanks to its high level of expressiveness, first-order logic is considered a standard way to represent the world, from mathematical problems to software specifications. These problems are translated into *logical sentences* [126] that can be either true or false regarding a specific context. However, it is important to note that first-order logic is *semidecidable* [86, 232], meaning there is no procedure capable of automatically deciding on the truth value of certain sentences.

1.1.1 Syntactic Definitions

First-order logic is a language, composed of words — also called *terms* — on which are built statements — the *formulas*. Together with the elements allowing words to be combined, a first-order logic language, denoted $\mathscr{L}$, can be defined as an alphabet composed of the four following disjoint sets (inspired by [90]):

- An infinitely countable set of variables $\mathscr{V}$, usually denoted by lowercase letters from the end of the alphabet, possibly indexed, such as x, x', x_1, y.
- A set regrouping propositional logic connectives, quantifiers, brackets, dots, and commas $\{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \exists, \forall, (,), .\}$, as well as the two specific symbols $\top$ and $\bot$. The precedence convention among logical operators is as follows: $\{\neg\} \prec \{\wedge, \vee\} \prec \{\exists, \forall\} \prec \{\Rightarrow, \Leftrightarrow\}$.
- A set of function symbols $\mathscr{S}_F$ in which each symbol is associated with its arity. Elements of this set are usually split between the *constants*, which are functions of arity 0 which use letters at the start of the alphabet a, a', a_1, b, and *functions*, denoted as usual with f, f', f_1, g. More formally, $\mathscr{S}_F$ can be defined as the union of the family $(\mathscr{S}_F^i)_{i\in\mathbb{N}}$ in which $\mathscr{S}_F^k$ represents a symbol of arity k, i.e., $\mathscr{S}_F = \bigcup_{n\in\mathbb{N}} \mathscr{S}_F^n$.
- A set of relational symbols and their arity $\mathscr{S}_P$, also called *predicates*, defined similarly as $\mathscr{S}_F$. They are denoted by uppercase letters such as P, Q, and R.

From these sets derives the notion of *term*, on which are built *formulas*, often denoted by the uppercase letters F and G, possibly indexed. The set of terms $\mathscr{T}$ is the smallest subset of words over $\mathscr{L}$, which contains constants and variables and is stable by associating any n-uple (for $n \geq 1$) of words to be the arguments of a function symbol of $\mathscr{S}_F^n$.

Definition 1.1: Term

The set of terms of first-order logic $\mathscr{T}$ over $\mathscr{L}$ is recursively defined by the family $(\mathscr{T}_n)_{n\in\mathbb{N}}$:

- $\mathscr{T}_0 = \mathscr{V} \cup \mathscr{S}_F^0$ where $\mathscr{S}_F^0$ denotes the functions of arity 0.
- $\mathscr{T}_n = \mathscr{T}_{n-1} \cup \left\{ f(t_1, \dots, t_k) \mid k \in \mathbb{N} \wedge f \in \mathscr{S}_F^k \wedge t_1, \dots, t_k \in \mathscr{T}_{n-1} \right\}$.
- $\mathscr{T} = \bigcup_{n\in\mathbb{N}} \mathscr{T}_n$

Atomic formulas also play the role of words in a logical sentence. The collection of atomic formulas consists of predicates and the two special symbols $\top$ and $\bot$. By connecting atomics formulas with connectors, the sentences, or *formulas*, of this language can be defined.

Definition 1.2: Formula

The set of formulas of first-order logic $\mathscr{F}$ over $\mathscr{L}$ is recursively defined by the family $(\mathscr{F}_n)_{n\in\mathbb{N}}$:

- $\mathscr{F}_0$ is the set of atomics formulas, i.e., the word W such that either (i) there exists $n \in \mathbb{N}^*$, an n-ary relational symbol R of $\mathscr{S}_P$ and n terms of $\mathscr{L}$ $t_1, \dots, t_n$ such that $w = R(t_1, \dots, t_n)$ or (ii) W is $\top$ or $\bot$.
- $\mathscr{F}_n = \mathscr{F}_{n-1} \cup \{\neg F \mid F \in \mathscr{F}_{n-1}\} \cup \{F_1 \oplus F_2 \mid F_1, F_2 \in \mathscr{F}_{n-1}\} \cup \{Qx\ F \mid F \in \mathscr{F}_{n-1}\}$ where $\oplus \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, $Q \in \{\forall, \exists\}$ and $x \in \mathscr{V}$.
- $\mathscr{F} = \bigcup_{n\in\mathbb{N}} \mathscr{F}_n$.

Formulas will be denoted by capital or Greek letters such as A, B, F, G, ϕ, etc. For the quantifier case, the vector notation $\forall \vec{x}$ and $\exists \vec{x}$ represents an indistinct set of quantified variables. The *dot notation* $[\forall, \exists]x.\ F$ allows the quantification to hold for the rest of the formula, i.e., t is equivalent to $\forall x(F)$.

In addition, we can also extract a particular subset of formulas, the *literals*, composed of atomics formulas and their negation.

1.1.2 Free Variables and Substitutions

In a formula, variables can be quantified using an existential ($\exists$) or a universal ($\forall$) binder or can appear without being bound. Throughout this thesis, two types of variables are distinguished: the *bound* variables, denoted as defined in Section 1.1.1 by letters such as x, x', x_1, y, and the *free* variables, which use the same letters but capitalized, i.e., X, X', X_1, Y. The set of unbound variables of a formula F is called the *free variables* of F and denoted $\mathrm{FV}(F)$. Formulas containing free variables can be syntactically manipulated to perform a *substitution*, which replaces them by terms following a mapping usually denoted σ. Last, a formula can be devoid of free variable and is subsequently called a *sentence* or said to be *closed*, and *open* otherwise.

Definition 1.3: Term Substitution

A substitution $\sigma : \mathscr{V} \to \mathscr{T}$ is a function from variables to terms and its application over a term t is denoted $\sigma(t)$. The domain of the function is the set $\mathrm{dom}(\sigma) = \{x \in \mathscr{V} \mid \sigma(x) \neq x\}$. The image of the function is the set $\mathrm{img}(\sigma) = \{\sigma(x) \mid x \in \mathrm{dom}(x)\}$. Thus, a substitution is defined by induction over t as follows:

- If t is a variable x then there are two cases:
  - if $x \in \mathrm{dom}(\sigma)$, then $\sigma(t) = \sigma(x)$;
  - else, $\sigma(t) = x$.
- If t is an n-ary function $f(t_1, \ldots, t_n)$ then $\sigma(t) = f(\sigma(t_1), \ldots, \sigma(t_n))$.

Having established the definition of substitution for a single term, the definition can now be extended to encompass an entire formula.

Definition 1.4: Formula Substitution

A substitution $\sigma : \mathscr{V} \to \mathscr{T}$ over a formula F is denoted $\sigma(F)$ and defined by structural induction over F:

- If F is an n-ary predicate $P(t_1, \ldots, t_n)$ then $\sigma(F) = P(\sigma(t_1), \ldots, \sigma(t_n))$.
- $\sigma(\top) = \top$ and $\sigma(\bot) = \bot$.
- If $F = \neg G$ then $\sigma(F) = \neg(\sigma(G))$.
- If F is a binary formula with the connectives $\oplus \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$ $F_1 \ \oplus \ F_2$ then $\sigma(F) = (\sigma(F_1)) \ \alpha \ (\sigma(F_2))$.
- If $x \in \mathscr{V}$ and F is a formula quantified by $Q \in \{\forall, \exists\}$ $Qx.\, G$ then $\sigma(F) = Qx'.\, \sigma(G[x \mapsto x'])$ with $x' \notin \mathrm{dom}(\sigma) \cup \bigcup_{t \in \mathrm{img}(\sigma)} \mathrm{FV}(t) \cup (\mathrm{FV}(G) \setminus \{x\})$.

The assignment of a variable X to a term t is denoted $X \mapsto t$. Thus, a substitution that maps X to a and Y to b is denoted $\sigma = \{X \mapsto a, Y \mapsto b\}$. Using substitutions, two formulas can become syntactically equal through instantiating their free variable(s) with the same term(s). As such, if for two formulas F and G there exists a substitution σ such that $\sigma(F) = \sigma(G)$, these two formulas are said *unifiable* with σ being their *unifier*.

Furthermore, for every pair of unifiable formulas, there exists a *most general unifier* θ (modulo renaming), denoted *mgu*. θ has the property that for every unifier τ of F and G, there exists τ' such that $\tau = \tau' \circ \theta$.

Furthermore, two formulas are said α-equivalent if they are syntactically identical when renaming the bound variables "properly". For example, $\forall x.\, P(x)$ is α-equivalent to $\forall y.\, P(y)$, but $\forall x.\, Q(x, y)$ and $\forall y.\, Q(y, y)$ are *not* α-equivalent, as y is free in the first formula and not in the second.

1.1.3 Semantic and Truth Value of a Formula

The end goal of writing formulas is twofold — express the world through a universal and logical language and reason over the sentences of this language. In order to do this, we must define the *truth value* or *logical value* of a formula, which can vary based on the conditions established within the particular *environment* and *interpretation* in which it is evaluated.

An interpretation contains information about the predicates and the terms for which they hold. An environment enhances an interpretation by giving a value to free variables. Then, it is possible to evaluate the value of a formula for a given environment.

Definition 1.5: Interpretation of formulas

Let $\mathcal{I}$ be an interpretation of $\mathscr{L}$, F a formula of $\mathscr{L}$, $\mathcal{D}$ the *domain* of $\mathcal{I}$ and e an environment.
An interpretation is a function that takes an element of $\mathscr{L}$ and an environment and returns an element of $\{0,1\}$, 0 meaning that the formula does not hold in these conditions, and does otherwise. In particular, the interpretation of F in e is denoted $\text{Val}_{\mathcal{I}}(F,e)$ and defined as follows:

1. $\text{Val}_{\mathcal{I}}(\bot,e)=0$.
2. $\text{Val}_{\mathcal{I}}(\top,e)=1$.
3. $\text{Val}_{\mathcal{I}}(R(t_1,\dots,t_n),e)=1$ iff $(\text{Val}_{\mathcal{I}}(t_1),\dots,\text{Val}_{\mathcal{I}}(t_n))\in R_I$.
4. $\text{Val}_{\mathcal{I}}(\neg F,e)=1-\text{Val}_{\mathcal{I}}(F,e)$.
5. $\text{Val}_{\mathcal{I}}(F_1\wedge F_2,e)=\text{Val}_{\mathcal{I}}(F_1,e)\times\text{Val}_{\mathcal{I}}(F_2,e)$.
6. $\text{Val}_{\mathcal{I}}(F_1\vee F_2,e)=max(\text{Val}_{\mathcal{I}}(F_1,e),\text{Val}_{\mathcal{I}}(F_2,e))$.
7. $\text{Val}_{\mathcal{I}}(F_1\to F_2,e)=max(1-\text{Val}_{\mathcal{I}}(F_1,e),\text{Val}_{\mathcal{I}}(F_2))$.
8. $\text{Val}_{\mathcal{I}}(\forall x\ F,e)=1$ iff for all $a\in\mathcal{D}$, $\text{Val}_{\mathcal{I}}(F,e[x:=a])=1$.
9. $\text{Val}_{\mathcal{I}}(\exists x\ F,e)=1$ iff there exists $a\in\mathcal{D}$ such that $\text{Val}_{\mathcal{I}}(F,e[x:=a])=1$.

Thanks to this definition, the semantics of a formula can now be deduced. This value can fluctuate depending on the environment, hence the need to define the notion of a *model* (resp. a *counter-model*) for a sentence, which is an interpretation that makes the formula true (resp. false) for all environment. For the rest of this thesis, we will focus on the value of a formula for any interpretation and then on the notion of *logical theorem*, usually shortened to *theorem*. A formula F is a theorem if every interpretation of F is a model. Conversely, a formula is said to be *unsatisfiable* if every interpretation is a counter-model.

Recall that logic is used as a way to describe and manipulate the world. Now that this language has been defined, we can expand toward *logical reasoning*, by utilizing

this logic in a specific reasoning method: the *method of analytic tableaux*.

1.2 Method of Analytic Tableaux

Although clausal proof techniques [11, 96, 97, 186, 209] have enjoyed success in automated theorem proving [225], some applications benefit from reasoning on unaltered formulas (rather than Skolemized clauses), while others require the production of proofs. Thanks to its equivalence to Gentzen's sequent calculus [134], these roles are fulfilled by provers based on the tableau method [166], as initially designed by Beth and Hintikka [41, 148], and later extended by Fitting [124] to be usable in automated deduction.

1.2.1 Free-Variable Tableaux Calculus

The tableau method proceeds by refutation, i.e., we take the negation of the formula and try to prove its unsatisfiability, and thus that the original formula is valid. A *tableau* (T, σ) for a formula F is a pair of a tree rooted in F, whose nodes are decorated with a set of formulas, and a substitution σ over the free variables of these formulas. A tableau is either (i) a single-node tree with the empty substitution, or (ii) obtained by application of an *inference rule* R on a tableau (T', σ'). These rules are part of a (refutationally complete) set of inference rules (Figure 1.1), and can be applied to a formula f to generate a new formula f', denoted $f \rightsquigarrow_R f'$. Thus, R can be one of the following:

- R is a unary (α, γ or δ) rule with premise P and conclusion C, and T is obtained from T' by adding C to a leaf of a branch that contains P,
- R is a n-ary (β) rule with premise P and conclusion $C_{i_{(1\leq i\leq n)}}$, and T is obtained from T' by extending a branch that contains P with n nodes decorated with $\{C_{i_{(1\leq i\leq n)}}\}$,
- R is a closure rule that extends the substitution σ'.

Note that in this setting, proof trees are n-ary, as only β-rules create new nodes. A branch in a tableau (T, σ) is said to be *closed* if it contains two literals L and L' such that $\sigma(L) = \sigma(\neg L')$, otherwise it is *open*. The set of open branches of T is denoted $\text{open}_\sigma(T)$. The tableau is closed if $\text{open}_\sigma(T) = \emptyset$. The notion of inference rule can be extended to tableaux themselves: thus, passing from a tableau (T, σ) to a tableau (T', σ') thanks to an inference rule R is denoted $T \hookrightarrow_R T'$. Finally, this derivation is called the *method of analytic tableaux*.

The process of applying rules on a tableau T rooted in a formula F to close the branches is called a *proof search* for F. The specification of the rules used, the order they are applied, or other parameters define the *proof-search procedure*. In the end, a *proof* for a formula F is a closed tableau. A proof-search procedure is said to be *fair* if each and every available rule has been applied at least once while no proof is found.

$$\frac{\bot}{\odot}\,\odot_{\bot} \qquad \frac{\neg\top}{\odot}\,\odot_{\neg\top} \qquad \frac{F, \neg G}{\odot_\sigma}\,\odot_\sigma \quad \text{where } \sigma(F) = \sigma(G)$$

(a) Closure rules.

$$\frac{\neg\neg F}{F}\,\alpha_{\neg\neg} \qquad \frac{F \wedge G}{\begin{array}{l}F\\G\end{array}}\,\alpha_{\wedge} \qquad \frac{\neg(F \vee G)}{\begin{array}{l}\neg F\\\neg G\end{array}}\,\alpha_{\neg\vee} \qquad \frac{\neg(F \Rightarrow G)}{\begin{array}{l}F\\\neg G\end{array}}\,\alpha_{\neg\Rightarrow}$$

(b) α-rules.

$$\frac{F \vee G}{F \quad G}\,\beta_{\vee} \qquad \frac{\neg(F \wedge G)}{\neg F \quad \neg G}\,\beta_{\neg\wedge} \qquad \frac{F \Rightarrow G}{\neg F \quad G}\,\beta_{\Rightarrow}$$

$$\frac{F \Leftrightarrow G}{\begin{array}{ll}F & \neg F\\G & \neg G\end{array}}\,\beta_{\Leftrightarrow} \qquad \frac{\neg(F \Leftrightarrow G)}{\begin{array}{ll}F & \neg F\\\neg G & G\end{array}}\,\beta_{\neg\Leftrightarrow}$$

(c) β-rules.

$$\frac{\exists x.\, F}{F[x \mapsto sko(Y_1, \ldots, Y_n)]}\,\delta_{\exists} \qquad \frac{\neg\forall x.\, F}{\neg F[x \mapsto sko(Y_1, \ldots, Y_n)]}\,\delta_{\neg\forall}$$

Where sko is a fresh (e.g., never previously used in the proof) Skolem symbol and $Y_1, \ldots, Y_n$ are the free variables in the branch.

(d) δ-rules.

$$\frac{\forall x.\, F}{F[x \mapsto X]}\,\gamma_{\forall} \qquad \frac{\neg\exists x.\, F}{\neg F[x \mapsto X]}\,\gamma_{\neg\exists}$$

Where X does not appear elsewhere in the proof.

(e) γ-rules.

Figure 1.1: Free-variable tableau rules.

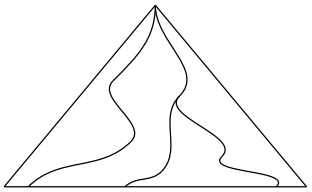
Definition 1.6: Fairness

A proof-search procedure is *fair* if and only if each formula on which a non-γ-rule can be applied occurs in a subsequent step, and every γ-rule will be computed an arbitrary number of times. More formally, for each formula f part of a tableau T:

- Every *closure*, α-, β-, δ-formula occurrence in T eventually has the appropriate tableau expansion rule applied to it, on each branch on which it occurs.
- Every γ-formula occurrence in T has the corresponding rule applied to it arbitrarily often, on each branch on which it occurs.

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg P(a)}{P(b)}}{\dfrac{\forall x.\, P(x) \vee (\forall y\, \neg P(y) \vee \forall z\, \neg P(z))}{P(X) \vee (\forall y\, \neg P(y) \vee \forall z\, \neg P(z))}\,\gamma_\forall}}{\dfrac{P(X)}{\sigma = \{X \mapsto a\}}\,\odot_\sigma \qquad \dfrac{\forall y\, \neg P(y) \vee \forall z\, \neg P(z)}{\dfrac{\forall y\, \neg P(y)}{\neg P(Y)}\,\gamma_\forall \quad \dfrac{\forall z\, \neg P(z)}{\neg P(Z)}\,\gamma_\forall}\,\beta_\vee}\,\beta_\vee}{\sigma' = \{Y \mapsto b\}\,\odot_\sigma \qquad \sigma'' = \{Z \mapsto b\}\,\odot_\sigma}$$

(a) Tableau — focus on formulas.



(b) Tableau — focus on structure.

Figure 1.2: Two ways to represent a tableau.

This notion is a requirement for one of the two main characteristics of a tableau calculus: *completeness*. It guarantees that if a proof can be found, the procedure will eventually find it. The other one, the *soundness* of the calculus, ensures that it only produces legal proofs.

Definition 1.7: Soundness

> A proof-search procedure $\mathcal{P}$ is *sound* when if $\mathcal{P}$ finds a proof for a formula F, then F is valid.

Definition 1.8: Completeness

> A proof-search procedure is *complete* when if F is valid then $\mathcal{P}$ finds a proof of F.

Throughout this thesis, a tableau (or a proof-search tableau) will be represented either as in Figure 1.2a, to highlight its formulas, or as in Figure 1.2b, to focus on the general structure of the tree. Unlike sequent calculus, formulas are not duplicated at each step, and the root of a tableau is at its top.

In addition, the special symbol $\partial(F)$ denotes the *delay* of the treatment of a formula F in the calculus. In other words, a *delay* is an interference in the regular rule application order of the proof-search procedure. The delayed formula is hidden from the proof search, waiting to become computable. It allows for a formula to be processed out of its usual place, i.e., after (resp. before) those with a higher (resp. lower) priority.

1.2.2 Terminology and Optimizations

The tableau method presented in the previous section is an enhancement of the original (*ground*) version of tableaux as first introduced by Beth and Hintikka [41, 148]. The main difference lies in the lack of free variables, impacting directly the instantiation rule: in the ground version, a variable has to be instantiated by a ground term.

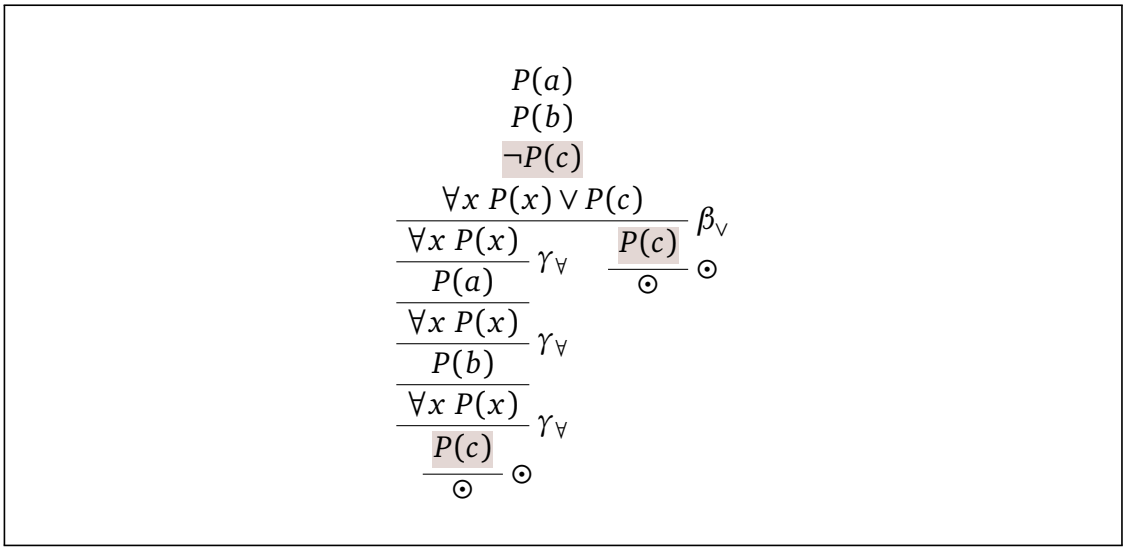


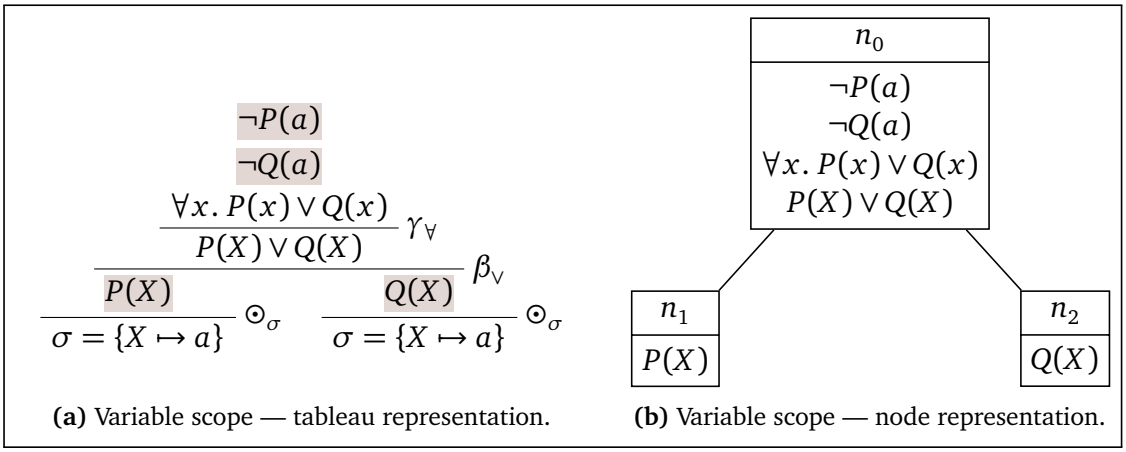
Figure 1.3: Inefficient instantiation (with free variables).

Ground tableaux provide an elegant way to *present* proofs, but its calculus is not suitable to efficiently *search* for one. In practice, considering that the free variable can be instantiated by a theoretically infinite number of terms, trying to guess the "right" term can be quite inefficient. For example, in Figure 1.3, the formula $\forall x\ P(x)$ instantiates x twice by the wrong terms (a and b) before trying c. This mechanism can be extended to a larger set of constants, resulting in a long chain of unsuccessful instantiations. [124] overcomes this issue by introducing the concept of *free-variable tableaux*, which allows delaying the instantiation of free variables, which are now used as a placeholder waiting for an instantiation. In this way, formulas become unifiable and the search for instantiation candidates is facilitated.

Scope of Free Variables and Substitutions The notion of the *scope* of a free variable for a node n in a proof tree defines the free variables to which n has access. For n, a free variable is called *local* if it was introduced by a formula F and there is no branching rule between F and n, and *non-local* otherwise. In Figure 1.4, the free variable X is considered as local for n_0, since it is introduced by $\forall x.\ P(x) \vee Q(x)$, and *non-local* after the disjunction, i.e., for n_1 and n_2. A free variable can also have multiple *occurrences* if the gamma that has generated it has been applied multiple times.

A substitution is said to be *local* to a node n if it only maps local variables, *non-local* otherwise. In case of a substitution closing the whole tree, this substitution is called *global*. An *eager closure* denotes the action for a branch to find a closure before all the rules have been applied.

Skolemization New challenges arise from free variables being part of the proof-search tree. The first one is the *consistency* of free variables between different branches of the tree, i.e., if a free variable X is substituted by a in a branch, it has to be substituted in the whole tree. The other one is directly related to δ-rules. Indeed, with free variables in the tree, it becomes harder to instantiate new Skolem symbols on the fly,



(a) Variable scope — tableau representation. **(b)** Variable scope — node representation.

Figure 1.4: The scope of a variable in free-variable tableau.

since the "final value" of the free variable is not yet known. To address this issue, the standard solution is the use of *Skolemization*, which uses a fresh *Skolem symbol*, i.e., a function symbol taking a set of free variables as parameters. In the case of a branch without free variables, the Skolemization function results in a constant.

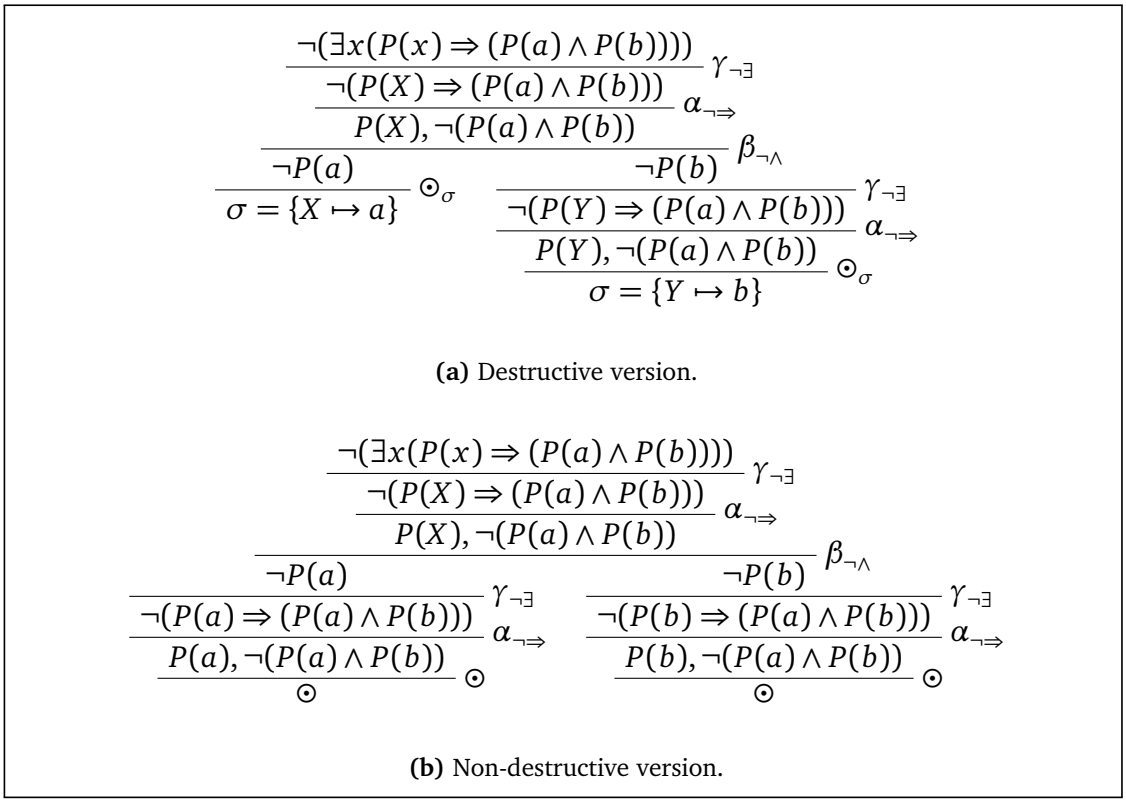
Definition 1.9: Skolemization

Let $\mathscr{S}$ be a subset of $\mathscr{F}$ and FV' a subset of $\mathrm{FV}(S)$. Let $F \in \mathscr{S}$ be such that F is of the form $\exists x\ F'$. The Skolemization function $s : (\mathscr{F}, T) \to \mathscr{F}$ of a formula is defined as follows: $s(F) = F'_{[x \mapsto f(Y_1,\ldots,Y_n)]}$ where $\mathrm{FV}' = \{Y_1, \ldots, Y_n\}$ and f is a *fresh* function symbol not occurring anywhere else in $\mathscr{S}$.

Skolemization is one of the key mechanisms of free-variable tableaux. Intuitively speaking, FV' correspond to a subset of free variables of $\mathrm{FV}(S)$ that can fluctuate according to the chosen Skolemization strategy. Those strategies vary in terms of the free variables used as parameters for the Skolem symbol (δ-, δ^+-, δ^*-rules [8, 144, 218]), and the Skolem symbol itself, which can be reused if introduced by the same formula more than once (δ^{+^+} and δ^{*^*}-rule [36, 81]). More optimized strategies also involve the use of ϵ-term (δ^ϵ-rule [137]), which is a meta-term that gives a constant that satisfies the Skolemized formula.

Destructive and Non-Destructive Versions By instantiating a free variable, applying a substitution affects the whole tableau. This step is said to be *destructive* because it may alter open branches sharing variables. Despite the replacement of free variables, the *destructive* version of tableaux maintains the *proof confluency* of the proof-search procedure. A calculus is said to be *proof confluent* [142] if it can derive a proof for any unsatisfiable set of formulas and from any given tableau, provided that the tableau itself has been generated using the calculus' rules. In simpler terms, a confluent tableau calculus never gets stuck in situations where the sequence of rules leads to a dead end.

Conversely, the *non-destructive* version of tableaux does not replace free variables, but instead reintroduces a specific γ-formula and instantiates it by a term previously



(a) Destructive version.

(b) Non-destructive version.

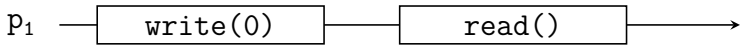
Figure 1.5: Proof of $\exists x(P(x) \Rightarrow (P(a) \wedge P(b)))$.

found by a substitution. The differences between these two versions are illustrated in Figure 1.5, where Figure 1.5a is a proof in a destructive tableau calculus and Figure 1.5b a non-destructive proof.

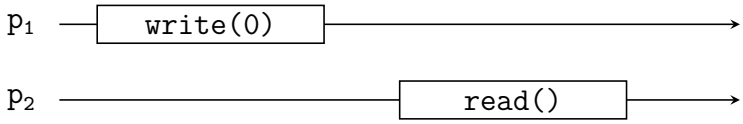
1.3 Concurrent Algorithmics

The widespread adoption of multi-process systems in all aspects of computer sciences has led to significant advancements as well as new challenges. These architectures enable better handling of the system resources and open a way for improved implementation of concurrent algorithms. However, they encounter specific issues, particularly when coordination is required. When multiple processes collaborate on a shared task, there is often a need to synchronize their actions and keep them informed about the overall task's progress. Without such coordination, certain processes might redundantly repeat work already completed by others or even attempt to operate on data that is no longer available.

This section introduces standard concepts of concurrent algorithmic, inspired by [140], and the main challenges it faces in Section 1.3.1. Subsequently, it dives into the topic of communication and memory management in Section 1.3.2 and concludes in Section 1.3.3 by introducing a semantic to represent a concurrent execution.



(a) Sequential execution of two operations on a resource R by the process p_1.



(b) Concurrent execution of two operations on a shared resource R: `write(0)` by p_1 and `read()` by p_2.

Figure 1.6: Sequential vs. concurrent executions of two operations, `write(0)` and `read()`, on one resource.

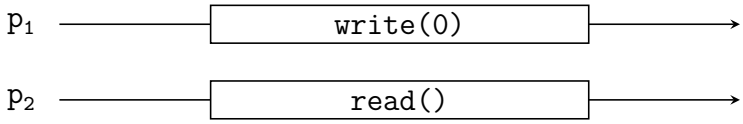


Figure 1.7: Concurrent accesses to a shared resource R that can lead to undesired behaviors.

1.3.1 Challenges of Multi-Process Architectures

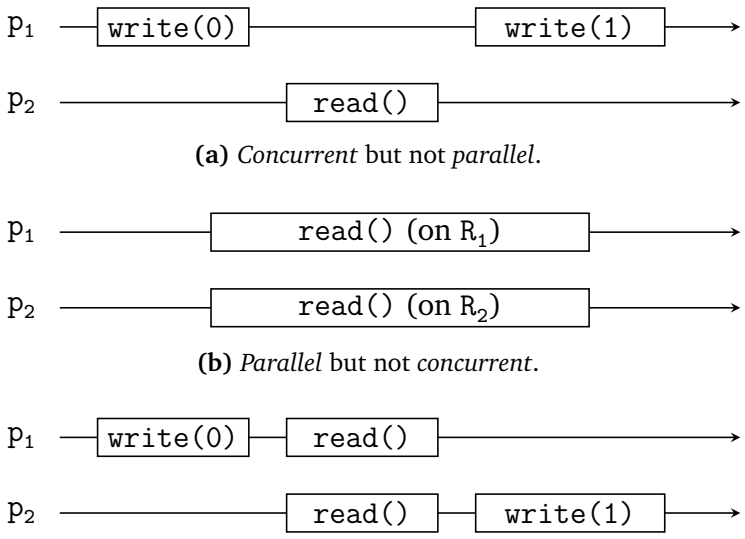
We consider a system that consists of a finite set of *n processes*, denoted by lowercase letters such that p, q, and r, possibly indexed, i.e., p_i. In the following, we use the notion of process to define indistinctly any type of concurrent unit (processes, threads, ...). Beyond accessing local variables, processes can *execute* an *operation* on *shared objects* or *resources*, such as reading from or writing to a variable. It is through such objects that processes achieve *synchronization* in their computations.

In a single process, operations are performed one after the other in a *sequential* way, as presented in Figure 1.6a. Let us consider a resource R, on which processes can perform two operations: `write(x)`, which writes the value x in R, and `read()`, which returns the current value of R. In this figure, the process p_1 first executes `write(0)` and then `read()`.

However, when it comes to multiple processes, they may naturally want access to the same resource, as illustrated in Figure 1.6b. In this case, p_1 first alters the value of R by executing the operation `write(0)`, and thus p_2 retrieves this new value with the `read()`.

In this situation, and without further restrictions, things can quickly get out of control. For instance, in Figure 1.7, both p_1 and p_2 attempt simultaneous accesses to R, aiming to respectively execute the `write(0)` and `read()` operations. This can lead to an incoherent behavior, in which either the R cannot be updated, or an erroneous value is returned after the reading operation. This management of shared data is the basis of all the concurrent algorithmic challenges [146].

To be able to properly deal with a large number of processes, sequential algorithms must be completely redesigned to take into account the concurrent accesses to shared data. In particular, synchronization mechanisms need to be implemented in order to prevent inconsistencies. Moreover, multi-process approaches encounter challenges such as deadlock [88] and the need to manage synchronous or asyn-



(a) *Concurrent* but not *parallel*.

(b) *Parallel* but not *concurrent*.

(c) *Concurrent* and *parallel*.

Figure 1.8: Distinction between *concurrency* and *parallelism*.

chronous behaviors [176].

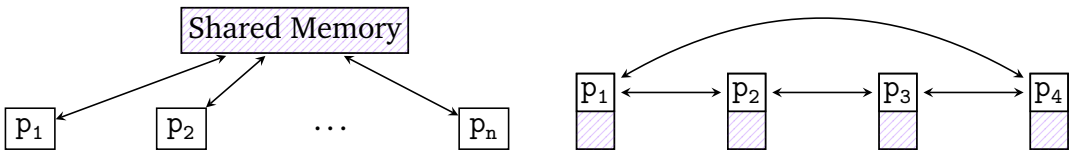
Before going further, it is important to make a distinction between the concepts of *concurrency* and *parallelism*. Although these two concepts are closely intertwined, the former concerns the structure of a program, whereas the latter refers to its execution. Figure 1.8 illustrates and distinguishes these two concepts. The scenario involves a system with two processes, $\mathtt{p}_1$ and $\mathtt{p}_2$, which can perform the two previous operations.

Let us begin with the setup presented in Figure 1.8a. In this configuration, each process performs an operation on the resource `R` one after the other, such that there is only one process accessing `R` at a given time. This configuration is said to be *concurrent* due to the simultaneous accesses of the same resource by two processes, but not *parallel* because the accesses did not occur at the same time.

In the system described in Figure 1.8b, each process accesses a distinct resource, $\mathtt{R}_1$ for $\mathtt{p}_1$ and $\mathtt{R}_2$ for $\mathtt{p}_2$. Since there is no shared resource, the processes can work at the same time, making this configuration *parallel* but not *concurrent*.

Conversely, in Figure 1.8c, the resource `R` is shared between all the processes. Thus, since the `read()` operation could not cause any malfunction in the system, it can be performed by the two processes at the same time, making this configuration *parallel* and *concurrent*.

In the literature of automated deduction, the term *parallelism* is often used as a general term to design the *parallelization of an algorithm*, which indiscriminately combines both *parallelism* and *concurrency*. Therefore, for the remainder of this thesis, we will adopt this interpretation of parallelization and clarify its usage when referring to parallel execution.



(a) *Shared memory*. **(b)** *Message exchanges*.

Figure 1.9: Comparison of memory management between *shared memory* and *message exchanges*.

1.3.2 Communication Between Processes and Memory Management

One of the primary techniques to implement the synchronization between processes is to make them *communicate*, as first introduced in [149]. By being aware of what others are doing, they will be able to manage their own actions to maintain a coherent system. Thus, the ability to broadcast and receive information among different processes is a fundamental principle of concurrent programming. There are two common approaches to dealing with concurrent systems: employing a *shared memory*, where all processes have access to a single common memory, or restricting memory to individual processes and enabling information exchange through *message passing*.

In a *shared memory* architecture (Figure 1.9a), the main memory is shared among all processing elements within a single address space. This setup includes specific segments called *critical regions* that are safeguarded against concurrent accesses. This involves the use of mechanisms such as *mutual exclusion* [111], which permits only one process to access these regions, as well as the use of a *fair scheduler* [158], ensuring eventual access to the resource for every process and preventing starvation. This type of architecture is commonly used in computers with multiple processors or cores. The advantages of this approach are fast access to shared data for all processes and efficient memory utilization, allowing efficient work with an important amount of data. However, it also introduces challenges such as race conditions, which need to be addressed with locks, potentially leading to parallel slowdowns or deadlocks, making implementations error-prone.

Conversely, in *message exchanges* (Figure 1.9b), each process is envisioned as a separate entity, akin to a software agent, with its own address space. Communication is achieved through message passing, i.e., by sending and receiving information among processes. These processes can have a uniform behavior or, conversely, each might have a distinct role and available actions (sending, receiving, or both). This approach can be used in both single computers and distributed systems such as grids or clusters of computers, and can easily manage the memory limitation issue.

Recall that the goal of the use of concurrency in this context is the design of a procedure based on a tree structure in which branches are explored in a simultaneous way. Each branch carries its own set of data, although the vast majority is shared with others. Interactions between branches are limited to parent-child communications. Even though a shared-memory architecture could have been envisaged, due to a large amount of common data, the limited interactions between processes (i.e., one-to-one

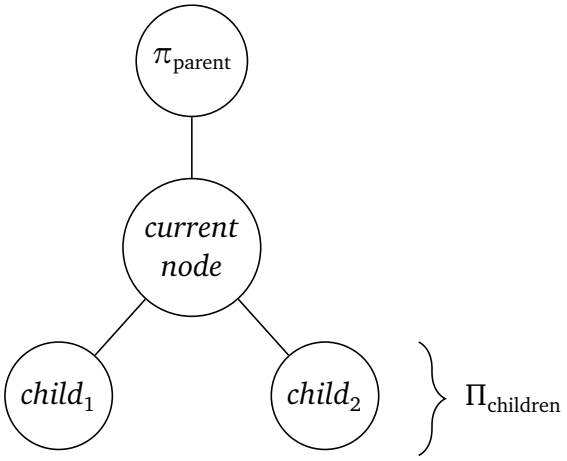


Figure 1.10: Node structure and local vision.

propagation, no broader broadcast) and the nature of these interactions have directed us toward the choice of message exchanges. This architecture also presents a large advantage when it comes to extension to distributed systems.

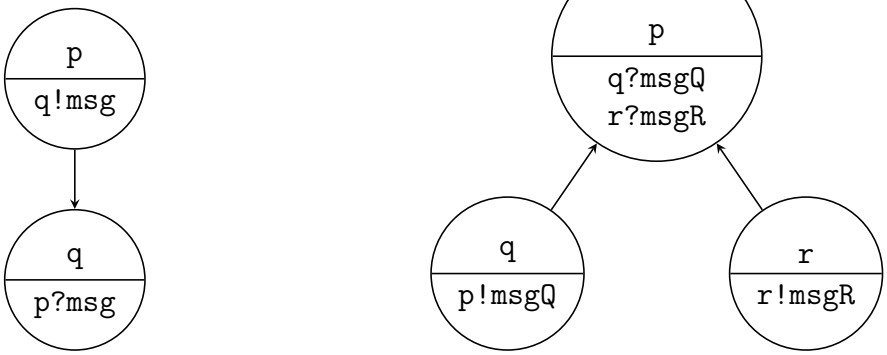
1.3.3 Semantic for Concurrency

In order to describe the behavior of concurrent processes, a dedicated semantics is needed. This section describes a simple WHILE language augmented with instructions for concurrency, in the style of CSP [149]. This semantic focuses on message exchanges and relies on strong parent-children communications.

In this language, each process has its own variable store, as well as a collection of process identifiers used for communication: π_{parent} denotes the identifier of a process's parent, while Π_{children} denotes the collection of identifiers of active children of that process. A node can have an arbitrary number of children, but only one parent. This representation (illustrated in Figure 1.10) describes the local view of a process, which has a special relationship with its parent and children.

Given a process identifier `p` and an expression e, the command `p!`e is used to send an asynchronous message with the value e to the process identified by `p`. Conversely, the command `p?`x blocks the execution until the process identified by `p` sends a message, which is stored in the variable x. Lastly, the instruction **start** creates a new process that executes a function with some given arguments, while the instruction **kill** interrupts the execution of a process according to its identifier. Let `p`, `q`, and `r` be tree processes. Thus, the following communication primitives, illustrated in Figure 1.11, are allowed:

- `p!msg`: the current process sends the message `msg` to `p`.
- `p?msg`: the current process is waiting for a message from `p`. Once received, this message is stored in the variable `msg`.
- **`start`** `fun`: a new process starts, running the function `fun`.



(a) p sends the message msg to q, which receives and stores it into the variable msg

(b) p receives the message msgQ from q (resp. msgR from r) and stores it into msgQ (resp. msqR).

Figure 1.11: Example of communication.

Procedure 1: $Proc_p$

```
   Data:
1  begin
2      for 1 to 10 do
3          start Proc_c
4      while ∃ π ∈ Π_children do
5          π ? v
6          if isEven(v) then
7              kill π
8      return 0
```

- `kill p`: the process p is killed.

An example of a concurrent algorithm written with this semantic is given in Procedure 1 and Procedure 2. The idea of this procedure is to create ten children that choose an arbitrary number and to kill them when they send an even one. In detail, $Proc_p$ starts by creating ten processes running the procedure $Proc_c$ (Line 3). The new processes executing $Proc_c$ choose randomly a number (Line 3) and send it to the parent process (Line 4) over and over again. Since the **!** operation is blocking, a child is put on hold until its message has been read by the parent. In the meantime, the parent process waits for answers from its ten children (Lines 4-5) and stores them into the variable v. If v is even, the corresponding child process is killed (Line 7), which is automatically removed from Π_{children}. Otherwise, the parent will later receive another message from this child, until the number becomes even.

This notation is used to represent the concurrent procedures in Chapter 3.

Procedure 2: $Proc_c$

```
   Data:
1  begin
2      while true do
3          v ← random()
4          π_parent ! v
5      return 0
```

Chapter 2
State of the Art

Contents

This chapter presents the state of the art of the different notions discussed in this thesis, allowing contributions to be positioned in relation to existing work. In particular, it details variations of the tableau methods and delves into optimizations and properties of the calculus, as well as the completeness of some implementations. Then, it introduces other reasoning methods in automated deduction and categorizes automated theorem-proving strategies through the prism of parallelism. Finally, it presents theory management in tableaux, with a particular emphasis on equality reasoning.

2.1 Optimizations and Completeness in Tableaux

This section focuses on the variations of tableau-based proof-search procedures, which can lead to multiple closure behaviors. Then, it presents some completeness proofs of actually implemented procedures, which are rather rare in the literature. Indeed, in order to have a competitive approach, some provers choose to drop completeness to achieve better in-practice results [120].

2.1.1 Proof-Search Variations in Tableau-Based Methods

This part introduces some of the main variations and optimizations of a tableau-based proof-search procedure. They can work in a local way or involve a global vision of a tableau, to take advantage of information coming from the different branches.

Skolemization Strategies A key mechanism in first-order logic is *Skolemization* [221]. The usual Skolemization strategy, introduced by Fitting in [124], takes all the free variables in the branch and creates a *fresh* Skolem symbol parametrized by those variables. This rule is referred to as δ-rule or *outer* Skolemization.

Through the years, optimized strategies have been developed to deal with existentially quantified formulas. Most of the optimizations rely on a finer choice of the free variables involved in the mechanism. The δ^+-rule [144], also known as *inner* Skolemization, optimizes standard outer Skolemization by solely keeping the free variables of the Skolemized formula as arguments, which ensures a more *local* solution and an exponential gain in proof-search size over outer Skolemization. The δ^*-rule [8] relies on a selection function to keep only the relevant formulas without affecting the integrity of the branch.

Other strategies involve refinements on the Skolem symbol itself. The δ^{+^+}-rule [36], or *pre-inner* Skolemization, as its name suggests, builds over *inner* Skolemization and adds a restriction: the Skolem symbol yielded by applying a δ^{+^+}–rule can avoid the *freshness* condition if it has already been instantiated by an α-equivalent formula. The δ^{*^*}-rule [81] extends the δ^* by delaying the Skolemization until the end of the proof search, attempting to perform a *global* Skolemization [82] and reducing the number of Skolem symbols and of variables dependencies in the proofs.

More advanced strategies also involve the use of ϵ-term (δ^ϵ-rule [137]), which is a meta-term that gives a constant that satisfies the Skolemized formula.

Pruning A method commonly implemented in tableau-based theorem provers is *pruning* [188]. This technique aims to reduce the proof-search space as well as the size of the proof tree [167, 194].

The idea behind the mechanism is to use the whole tree to remove irrelevant or redundant subtrees. In practice, the procedure attempts to detect branches that are similar to already closed ones and to eliminate them by grafting the corresponding branch in place of the deleted one. More specifically, they compare different subtrees and attempt to show that one of them (together with its successors) is redundant in the presence of the other. A natural approach here is to exploit subsumption between tableaux, in a similar manner to the way subsumption between clauses is used in formula saturation procedures such as resolution. Such a pruning mechanism is implemented into tableau-based solvers such as Zenon [65] and Ramcet [79].

This mechanism can be extended even in the context of iterative deepening. In this case, previously locally closed branches are memorized [6]. However, this technique is very memory-consuming, and cannot be extended to global substitution or unsuccessful substitution attempts because the new bound can lead to new solutions. SETHEO [181], together with the algorithm described in [168], implements a variation of this method for connection tableau with a local failure cache mechanism.

2.1.2 Completeness of Proof-Search Procedures

Many texts on free-variable tableaux feature a discrepancy between calculi and proof search procedures. Indeed, the completeness of the calculus does not imply the

completeness of its adaptation by various proof-search procedures. Some tableau-based procedures, mainly working with a global closure rule, some procedures have proven to be complete.

In tableaux, a *global closure* refers to finding a substitution that closes the whole tableau and applies it at once, whereas the notion of *eager closure* embodies the mechanism of closing branches one after the other, propagating the substitutions to the other branches.

Global Closure One of the first procedures in this vein can be found in [124]. It describes an "MGU atomic closure rule" that corresponds to what we call eager closure, but later gives a proof search procedure that performs global closure. This procedure relies on a fair expansion of the tableau and iterative deepening where the closure of the whole tableau is tested when the γ-rule limit has been reached. This technique, while it has the merit of being one of the first, is not very effective in practice since it requires waiting until reaching the limit to test the closure.

Some approaches avoid eager closure and backtracking entirely. [136] proposes an incremental closure rule, which keeps track of substitutions for each branch and eventually closes the whole tableau at once. This technique is implemented in the prover Princess [211] and proved complete.

Clausal Tableaux In the context of clausal tableaux, [30] uses a notion of tableau subsumption to avoid fairness issues, together with reconstruction steps to get around the destructive nature of closure. [44] and [25] both provide proof-confluent calculi that avoid destructive instantiation and discuss the issue of fair proof search. $\mathcal{ME}(\mathsf{LIA})$ [26] provides a completeness proof of the calculus augmented by additional rules to handle linear arithmetic.

Compiled Tableaux Alternatively, [201] describes an approach to perform a proof search on a compiled proof tree. The basic idea is to compile a fully expanded tableau into a program that carries out the proof search at runtime. First, an arbitrary first-order formula is transformed into a graph representation of a fully expanded tableau for it. Then, the graph is compiled into a program that shows the formulas' inconsistency when it is executed. The execution reflects the proof search in semantic tableaux and tries to close every branch in the tableau. The advantage of this approach is that some of the efforts for the proof search (namely, expanding the tableau) can be moved to a preprocessing phase that derives the graph and generates the program for it. This results in a more efficient proof search since the tableau does not need to be expanded anymore.

Eager Closure In [94], a definition of the eager closure rule is given, which explicitly points out the risk of unfair proof search. It mentions tableaux enumeration with backtracking and iterative deepening as a solution but without an explicit procedure. [38] describes an implementation of a prover with eager closure, yet the completeness proof given is for a variant of that prover with global closure. The prover Zenon [65] adds the required instances to a branch instead of instantiating the free variables,

and implements a pruning mechanism to avoid duplicated work for branches that share variables. One of the reasons behind the lack of completeness proof with eager closure is that the mechanism is more related to the non-destructive version of tableaux, for which completeness is harder to achieve, as stated by Hähnle: "At the present time, no strongly complete, destructive tableau proof procedure is known that works well in practice" [208].

2.2 Parallelism and Concurrency in Automated Deduction

Automated reasoning aims to automatically produce a proof of mathematical theorems. The large search spaces explored through the proof search have motivated early parallelizing endeavors. Parallelization in automated deduction was first used between the end of the 1980s and the beginning of 1990 both for satisfiability in propositional logic (SAT) [1, 16, 172, 222] and first-order automated theorem proving [53, 57, 63, 106, 214, 229]. While the latter has known a drop in interest, research in the former continues nowadays [145, 157, 171], recent improvements have led to renewed interest in the first-order case, especially for *model-based* methods [60].

Usually, classifications of reasoning techniques split them according to their logical inference type [208], i.e., resolution or analytic rules, while others focus on the proof search itself [53, 54, 59]. In this thesis, as we focus on first-order classical logic, we present two main paradigms: the *resolution*-based methods and the *tableau*-based ones.

On the other side, parallelization in automated deduction can arise from all the layers of those techniques, from inference rule to apply to the proof-search plan itself. Whereas the former suggests the parallel exploration of the proof-search space, the latter can technically describe a theorem prover running different variations of the proof-search procedure at the same time [56, 59].

In this section, we focus on introducing reasoning techniques in first-order logic and distinguish the different approaches in parallel automated deduction. We start by categorizing theorem-proving strategies, and then we study the parallelization of these techniques, from the improvement of the actual methods with parallelism to parallelism by design strategies.

2.2.1 Theorem Proving Strategies for First-Order Logic

In automated theorem proving, the best-known methods are either *resolution*-based or *tableau*-based. Resolution-based methods transform the goal formula into a clausal normal form and then perform a sequence of derivation steps on this set of clauses using a saturation algorithm. In contrast, tableau-based methods work directly on the original formula by reducing goals into subgoals and attempting to resolve these subgoals. Both of them work by refutation, i.e., attempt to prove that the negation of the original formula is unsatisfiable (by generating the empty clause for the former, and closed tree for the latter).

The main differences come from the form of the initial statement, their adaptability to other theories or logics, and the output. Coarsely, the resolution mechanism can be seen as a system working on a base of facts that tries to generate new facts until reaching a given goal. Conversely, tableaux start with a goal and subdivide it into subgoals, applying inferences rules until reaching two contradictory formulas. Some hybrid methods also borrow attributes from both of the approaches. For example, the inverse method attempts to saturate a base fact such as resolution but works directly with formulas instead of clauses.

Resolution-Based Methods

Resolution-based methods rely on a saturation process, aiming to produce new elements. This term refers to a wide range of techniques based on the *resolution rule* and improved by multiple optimizations. Inspired by the ideas presented in [97], a refutationally complete calculus was introduced by Robinson in 1965 [209].

Resolution Rule The central element in resolution is the notion of *clause*, which is a finite disjunction of literals, viewed as a multiset. Variables can be part of a clause and are therefore interpreted as universally quantified. This method transforms the initial formula to prove into clausal form and proceeds iteratively by applying a *resolution rule* between two clauses which contains two complementary literals, generating a new clause called *resolvent*. The resolution rule is as follows, l meaning that the literal l is selected in the clause:

$$\frac{C_1 \vee l' \qquad C_2 \vee l}{\sigma(C_1 \vee C_2)}$$

where C_1 and C_2 are two clauses and σ is a *mgu* between l and l'. Since the rule attempts to find a resolvent between two clauses, it is called *binary resolution*. The calculus generates a chain of derivation until either (i) it generates the empty clauses, denoted $\square$, meaning the validity of the original formula or (ii) it saturates the facts base, leading to a failure. This calculus serves as a base for various automated theorem provers as Otter [177], Vampire [163], or Waldmeister [71], including the best state-of-the-art ones.

Improvements of the Original Method In order to improve the in-practice efficiency of the method, the original calculus has undergone numerous refinements and optimizations. This has led to *ordered resolution*, which endows the resolution calculus with an order over literals and a selection function [164], limiting the number of clauses that can be generated and reducing the search space.

Another improvement comes from the resolution rule itself. While *binary* resolution manages to find a resolvent between two clauses, *hyper-resolution* attempts to find one between an arbitrary number of clauses [210], but this approach has led to only a few implementations [192].

Equality Management: Paramodulation and Superposition The equality predicate plays a central role in many first-order logic problems. However, its addition in an axiomatic way to the problem is subject to the generation of too many (and most of the time useless) clauses, making it space and time-consuming. Conversely, *paramodulation* [186, 237] offers to handle it directly in the language itself, as a dedicated inference rule. Thus, the paramodulation calculus corresponds to the resolution calculus improved by the following rule:

$$\frac{C_1 \vee s \approx t \qquad C_2}{\sigma(C_1 \vee C_2[t]_p)}$$

where $C_{2_{|p}}$ is the subterm of C_2 at position p, and $C_2[t]_p$ denotes the result of replacing in C_2 this subterm by t. This method is mostly used by resolution-based theorem prover to deal with equality, and can also be implemented as a purely equational paradigm, simulating non-equational inferences via appropriate equality inferences, such as in the E [213] theorem prover.

Just as the original resolution, paramodulation can be improved by redundancy management techniques [169, 197, 219, 238] and further restrictions [187]. When used with a reduction ordering on terms, the resulting calculus is called *superposition* [10, 151, 159] and is implemented in provers such as Zipperposition [93] or Spass [236].

Tableau-Based Methods

For first-order logic, tableau-based methods were first designed to work with formulas and thus extended to the clausal case. However, tableaux is still the major reasoning method used in some non-classical logic, for which a clausal normal form is unknown.

Analytic Tableaux The method of first-order analytic tableaux and its variations, as introduced in Section 1.2, differs from resolution by reasoning on the original formula, allowing the output of a sequent-like proof. Multiple automated theorem provers based on this method or its variations have been developed: ${}_3T^AP$ [35], HARP [188], leanT^AP [38], tableau [92], or Zenon [65] (which implements the non-destructive version of tableaux) to name a few. An overview of tableau-based theorem provers is available in [215].

Regular tableaux are tableaux in which none of the branches contain more than one occurrence of the same formula [27]. They are designed to avoid redundancy, but their implementation requires attention to a few points, especially due to the definition of the α-rule (i.e., not applicable in case of the generation of multiple formulas) and its interaction with the closure rule which instantiates free variables.

Clause Tableaux and Refinements Tableaux can also make use of formulas in clausal form, resulting in *clause tableaux* [141, 143, 166]. Clause tableaux mainly constitute a syntactic simplification of full first-order tableaux, which makes the mechanism closer to resolution and thus more suitable for automated deduction.

Moreover, due to the uniform structure of formulas in clausal form, it is much easier to detect additional refinements and redundancy elimination techniques than for the full first-order format. The mechanism is thus reduced to three rules: the branching- (β-) expansion rule, the instantiation- (γ-) rule, and the closure rule.

Clause tableaux can follow the same refinement as resolution, i.e., by constraining the choice of applicable rules or using a selection function and a term ordering. These refinements can be applied to usual tableaux or clausal tableaux, leading to *connection tableaux*, pioneered by Andrews [3] and Bibel [42]. This variant of tableaux is closely related to *model elimination* [169], which relies entirely on a restriction on the rule application: the next selected clause needs to have at least one literal in common with the one in the parent node. This control mechanism aims to guide the proof, avoiding potentially useless steps. Those methods are currently the most implemented in tableau-based theorem provers, with CPTHEO [127], leanCoP [191], METEOR [5], PARTHENON [67], PARTHEO [216], and SETHEO [181] for instance. A variant of the connection calculus without clausal form [189] has also been developed, based on matrices, and implemented into nanoCoP [190].

Regular tableaux can also be adapted to the clausal case, i.e., when none of the branches contains more than one occurrence of the same literal. In parallel, the *n*-ary rule of *hyper-resolution* can also be applied to the tableau case, resulting in *hyper tableaux* [27].

The Inverse Method

Less known than its predecessors, the *Inverse Method* [101] lies somewhere in between, borrowing characteristics from both resolution-based reasoning techniques and tableau-based ones. Just as tableaux, it deals with first-order formulas, but instead of trying to reduce the goal into subgoals, it attempts to construct goals from previously proved subgoals. The inverse method was first designed by [134] and used to prove the decidability of intuitionistic propositional logic and then extended in [174, 175, 180].

This method is mostly used in the case of non-standard logics [234]. For example, [85] presents an automated theorem prover for first-order intuitionistic linear logic based on the inverse method. However, it has generally received little attention in terms of implementation.

2.2.2 Parallel Theorem Proving

Concurrency and parallelism in automated deduction have been widely studied over the years. Concurrency has been used as the basis of a generic framework to present various proof strategies [122], to facilitate cooperation between proof systems with complementary strengths [39], or to allow distributed calculations over a network [239]. A lot of research has also been carried out on the parallelization of proof search procedures [59]. This has led to a fine classification of the parallel behaviors [53] to distinguish the different parallelization levels: *parallelism at the term level*, *parallelism at the clause level*, and *parallelism at the search level*.

Parallelism at the term level means that multiple processes can attempt to access the same term or literal, for example, when distinct inferences can be applied

simultaneously to different literals of a clause.

Parallelism at the clause level refers to parallel access to distinct clauses or subgoals. For tableau-based theorem provers, this method implies that multiple processes progress on different branches or clauses, whereas a sequential version would have tested one after the other via backtracking. This kind of strategy is employed by some provers [5, 67, 216], and requires communication between processes. For resolution-based methods, the main idea is to parallelize the choice of a clause and update cooperatively the set of available clauses. The idea was implemented in Roo [170], a parallelization of Otter [177].

The first two categories are subject to a lot of conflicts, mainly due to concurrent access (for example, variable replacement or clause deletion). Based on these observations, the parallelization was directed toward the search level, pioneered by the Clause-Diffusion method [57].

Parallelization of the Proof Search

Parallelism at the search level alters the whole derivation, as multiple processes search in parallel for a proof. This type of parallelism involves *parallel search* and is characterized by communication among the processes. Parallel search yields *multi-search*, where different search plans are employed by the processes, and *distributed search*, where the search space is divided among all the processes. Both of these approaches can be combined, along with the other level of parallelism. This type of parallelism allows more advanced strategies and is responsible for most of the recent improvements of automated theorem provers.

The main idea of parallelization of the proof search is to generate n processes that search in parallel for a proof, each of them managing its own derivation and related data. In this configuration, the success of one process means the success of the entire proof search. Since each process manages its own data, the issue of conflicts of the previous levels disappears. While the two previous types of parallelization aim at speeding up a given search, parallelism at the search level aims at finding a proof sooner by searching in different ways. The counterpart of the per-process data management is the redundancy implied by the duplication of information. Another general issue with parallelism at the search level is the differentiation between the various processes' searches. Indeed, since the processes work from the same problem, their searches could be close, but having two processes performing the exact same search is useless. The idea is to minimize the overlap of the searches performed by the parallel processes [52, 55–57, 63].

Thus, two approaches have emerged: applying different search plans, of working on different data. This distinction was originally presented as *competition versus cooperation* [106, 214, 229], and then renamed into *multi-search* and *distributed search* [53, 56], highlighting their ability to be used together.

Multi-search

A *multi-search method* is a parallel search method within the proof-search processes differ in terms of *search plans*. In a broader context, multi-search may also be extended to processes applying different inference systems (such as two different resolution-based systems, or a combination of tableau-based and resolution-based systems). In multi-search with *homogeneous systems*, the processes have the same inference system but different search plans. In multi-search approaches with *heterogeneous systems*, the processes can differ either in the inference systems only or in search plans and inference systems.

In tableau-based systems, different search plans can arise from the (non-exhaustive) following elements: the limit for iterative deepening, the priority order on rules, the selection rule, or any combination of thereof. Some examples of tableau-based heterogeneous systems include CPTHEO [127], which launches SETHEO and the resolution-based prover Delta [217], and HPDS [227].

For resolution-based strategies, multi-search homogeneous systems were introduced by the *Team-Work* method [108]. The Team-Work method works with n processes and one supervisor. All processes have the same inference system, input problem, and time limit, but differ in search plans (for example, a different selection function [2]). Every process builds its own derivation and evaluates its progression at the end of the time limit. Then, the best one (w.r.t. the evaluation function) becomes the supervisor for the next round, in which all the processes restart their search, based on the derivation of the new supervisor. DISCOUNT [109] implements such an approach. Heterogeneous case includes the TECHS system [107], which runs DISCOUNT [109], SETHEO [181] and Spass [236] in parallel.

Distributed proof-search

A *distributed-search method* is a parallel search method within the *search space* is divided among the parallel proof-search processes. The initial data are split and each process works with its part. This division is made by knowing the type of rules that can be applied by the processes, hence this method is usually paired with homogeneous systems. Distributed search may also apply different search plans to the processes, leading to methods that combines distributed search and multi-search.

In tableau-based methods, this separation of the proof-search space often refers to branches. For instance, in HOT [161], a tableau-based automated prover for higher-order logic, each branch is managed independently and cooperates concurrently to build a proof after having fully developed (or closed) a branch.

For resolution-based methods, the *Clause-Diffusion* method [57] was a pioneer in the field, even being the first parallel-search method for automated first-order theorem proving [59]. In this method, all processes work with the same inference system and search plan, whereas the clauses are partitioned among them. This method pertains to the homogeneous category and was implemented in Aquarius [57, 61, 62], a parallelization of Otter [177], and Peers [64] provers. It was then extended by *Modified Clause-Diffusion* [58] into Peers-mcd [56].

2.3 Theory Reasoning in Tableaux

A theory represents knowledge from a given domain. In the realm of logic, it can be translated as a set of satisfiable sentences and may incorporate dedicated reasoning techniques. Whereas there exist efficient methods to deal with a specific theory, there is no universal approach to handle them all, and the addition of theory axioms into the problem hypotheses is rarely usable in practice. Indeed, some theories can have an infinite number of axioms, and the resulting increase in the problem size does not allow efficient reasoning, in addition to an incapacity to identify relevant axioms. Conversely, the use of domain-specific knowledge to develop efficient reasoning techniques gives better results and is currently the standard way to deal with a given theory. This involves the interaction between a general-purpose *foreground reasoner* and a specialized *background reasoner* designed for dealing with problems related to a particular theory. This collaboration occurs during key moments of proof search, guiding the process in the correct direction.

Following the pioneering work of Stickel on resolution [200, 224], theory reasoning methods have been described for various tableau-based calculi: path resolution [183], the connection method [29, 196], model elimination [23], connection tableaux [28, 29, 128], and the matrix method [184].

This section focuses on theory reasoning in first-order semantics tableaux. Due to its expressiveness and its prevalence in the problems, equality stands out as a particular theory. Handling equality reasoning in tableaux is notably challenging, leading to a wide range of studies and techniques. Thus, a particular emphasis is put on equality reasoning management in tableau-based methods in this section. Then, we shift to other theories and present the general management of theory reasoning in tableaux.

2.3.1 Equality Handling in Tableaux

There are two primary techniques for handling equality semantic tableaux: *partial equality reasoning*, the more straightforward method consisting of adding new rules to the tableau calculus, and *total equality reasoning*, which hinges on E-Unification, a decision procedure that determines branch closure without additional expansion rules.

Partial Equality Reasoning The earliest methods for incorporating equality into the grounded version of semantic tableaux emerged in the 1960s [154], following research on introducing equality into sequent calculi [156]. The idea of this method is to apply the equality axioms on all the available formulas in order to generate new terms and eventually find a closure. Thus, if a branch B contains a formula $P(a)$ and the equality $a \approx b$ holds, then the equality is "applied" on $P(a)$, generating $P(b)$ which is added to B. However, these newly introduced expansion rules, while based on grounded tableaux, present a significant drawback: they are symmetrical and lack constraints on their application, resulting in a considerable amount of non-determinism and an expansive search space, often leading to a multitude of irrelevant formulas. Continuing, if B contains the formulas $f(a) \approx a$ and $P(a)$, then all the formulas $P(f(a)), P(f(f(a))), \ldots$ can be added to B.

Enhancements to these equality expansion rules were proposed in [205], aimed at reducing the search space through strategic restrictions and a more goal-directed approach. These refined rules selectively use potentially complementary literals for expansion. [70] presented a set of rules implementing a completion procedure for grounded tableaux, but these rules are intricate and do not extend well to free-variable tableaux.

[124] extended this partial approach and adapted it to free-variable tableaux. The key difference lies in the fact that equality rule applications may involve free variables' instantiation, hence the need for unification. For instance, a branch containing the inequality $\neg(f(X) = f(a))$ is closed when the substitution $\{X \mapsto a\}$ is applied. However, this method faces the same challenges as the equality-free substitutions. If a "wrong" substitution or an equality rule is applied, the proof search can continue indefinitely until it reaches a γ-rule application point or initiates a backtracking mechanism. While restrictions can be applied to this calculus to reduce the number of used rules, this comes at the expense of completeness. Hence, applying equalities to equalities is essential for achieving a complete calculus, but it induces considerable noise and fills the proof-search space with a lot of unnecessary terms.

Efforts to transform more sophisticated and efficient methods, such as completion-based approaches, into simple tableau expansion rules face difficulties. The shared problem among these partial reasoning methods, based on extra tableau expansion rules, is that equalities can be applied without any restriction. The symmetry of these rules leads to extensive search space, making it challenging to solve even relatively straightforward problems in a reasonable amount of time [32]. However, some implementations of such partial equality reasoning methods have led to efficient results in practice [65].

Total Equality Reasoning In contrast, *total equality reasoning*, by avoiding the addition of equality expansion rules, transforms the task of finding a closing substitution in a tableau branch into solving an E-unification problem. This approach permits the use of various algorithms for E-unification problem-solving. Additionally, [33] demonstrated that E-unification-based methods significantly outperform those based in additional rules.

The main idea behind this concept is to define an E-unification problem comprising a pair of complementary literals s and t and an equality set E. Thus, by using the equality of the branch, we search for a substitution θ such that $\theta(E) \vdash \theta(s = t)$. Multiple algorithms solve the E-unification problem, contributing to the efficiency of this approach compared to the partial one. Moreover, different forms of E-unification exist based on the type of tableau (grounded, free variables, etc). For free-variable tableaux, the corresponding problem is known as the *rigid E-unification* problem.

In a tableau, solving an E-unification problem leads to close a single branch, which makes it related to eager closure. Conversely, closing multiple branches (up to the whole tableau) at once such as a global closure is achieved by *simultaneous* E-unification. In the simultaneous case, all the branches attempt to find a common solution to their own E-unification problems.

The decidability of a (simultaneous) rigid E-unification problem was a long-time

running problem. While [160] established the decidability of the non-simultaneous scenario and [129] its NP-completeness, the simultaneous case was ultimately proven to be undecidable [102].

The first attempt to define the generalized unification problem for addressing equality in rigid variable calculi can be found in [43]. Later, the importance of rigid E-unification for automated theorem proving was initially outlined in [130], which refines the former idea in [129] to suit equational reasoning in tableaux. A complete procedure was then developed in [103], which uses E-unification to address equality reasoning in tableau-based systems.

Despite the problem's inherent NP-completeness [131], meticulous attention has been dedicated to enhancing efficiency. The problem's exponential nature largely stems from the multitude of applicable rules and the checking of constraint satisfiability. To address the latter, techniques such as those outlined in [135] have been employed, leveraging backtracking and offering *constrained literals*. These constrained literals involve linking a constraint that comprises the unification and superposition rule steps necessary to derive a specific literal to the literal itself.

Recently, *simultaneous bounded rigid E-Unification* (BREU) [13–15] has emerged. BREU represents a novel variant of rigid E-unification characterized by boundedness, wherein variables exclusively represent terms from finite domains. This variant, although NP-Complete, is decidable, and holds promise as an efficient implementation candidate. The pursuit of even more efficient strategies for dealing with equality within tableau-based proof-search procedures is still ongoing.

2.3.2 Other Theories and General Theory Management

The usual way to deal with a specific theory is to call a background reasoner, which uses dedicated procedures for a given theory. This section classifies different types of background reasoners, with various degrees of embodiment into the calculus itself.

SMT Solvers

The current standard way to deal with a theory includes the use of a SMT solver. Satisfiability Modulo Theories (SMT) refers to the problem of determining whether a first-order formula is satisfiable with respect to some logical theory. Over the years, SMT solvers have proven to be the most efficient way to manage a theory [22]. The most famous examples of SMT solver are Z3 [99], CVC5 [17], Yices [118] or Alt-Ergo [89].

Provers and SMT solvers usually interact in the following way: the prover applies its deduction mechanism, which generates new instances, and calls the dedicated SMT solver to solve them. Then, the solver checks the satisfiability of the given instances, and if so, provides a model of this satisfaction. Following the answer, the prover can either continue with the current instantiations or, conversely, try a different one. This sequence of exchanges guides the proof search toward the final proof. Approaches such as the ones in Avatar [45, 206], Sledgehammer [46] and Spass+T [203] make use of SMT solver to reason with theories.

Historically, this driven-by-model approach often did not deal with quantification, which is managed by the foreground reasoner. More recently, state-of-the-art SMT solvers have been equipped with means to deal with quantification [98, 132, 133].

In the context of first-order semantics tableaux, calling a SMT solver can interfere with the proof's construction. Moreover, since SMT solvers work without quantification, free variables reintroduction can lead to potentially redundant calls, requiring particular attention to when the call to the background reasoner has to be made.

Theories Reasoning Integrated to the Calculus

Although SMT solver offers a modular way to handle theories, another approach involves integrating theory reasoning directly to the calculus. For instance, some provers have developed techniques to directly integrate Presburger arithmetic reasoning into the tableau calculus, designing new rules that intertwine with the original ones. This method offers a better handling of quantification and allows the production of a proof.

Zenon [77] integrates a decision procedure that relies on a Simplex and Branch and Bound approach [21]. In the same vein, Princess [211] incorporates a sequent calculus that combines ideas from free-variable constraint tableaux with the Omega quantifier elimination procedure [204], and is used as a background reasoner, for example, by the provers Elderica [150] and Tricera [119]. An approach to embed algebraic constraints in tableau calculi is also described in [198]. A little further away from our approach, the model-elimination prover $\mathcal{ME}$(LIA) [26] works with a calculus improved by a decision procedure for linear arithmetic.

Deduction Modulo Theory

Previous implementations offer an efficient way to deal with specific theories. However, with the increase and diversification of theories (for instance, those coming from industrial problems), the idea of an extensible background reasoner capable of handling a multitude of theories has gained traction. Although a general-purpose theory-handling background reasoner could not compete against a dedicated one, the ability to reason in a generic way is an interesting property. Moreover, the two approaches can be used in complementary ways.

This generic theory handling is the motivation behind *Deduction Modulo Theory* [115]. The main idea is to provide a generic framework that turns axioms into computational rules, by removing them from the formula to prove and trigger only the relevant ones. Similar approaches have already suggested the conversion of theory axioms into deduction rules: [202] has suggested a generation of introduction and elimination rules from axiom, and [199] has designed unification modulo associativity in a resolution system. More recently, superdeduction [69] provided an extension of [202] by compiling the introduction and elimination rules, performing the deduction steps only once, and providing more compact rules.

These approaches serve the common purpose of getting rid of axioms and facilitating the proof search. While the previous approaches try to include axioms in the deduction part of the system, Deduction modulo theory transforms axioms into computation steps. Initially, designed to deal with specific theories such as simple

type theory [114], arithmetic [117], and Zermelo's set theory [116], this approach was extended to be able to handle any axiomatized theory. The results yielded by this approach on Zenon [65] gave birth to the provers Super Zenon [153] and Zenon Modulo [104] and inspired iProver Modulo [72].

Incremental Theory Reasoning

In addition to considering the effectiveness of both the foreground and background reasoners, interactions between them assume a pivotal role in optimizing the efficiency of the integrated system. In particular, one of the key problems is to figure out the right moment to call on the background reasoner and the resources to allocate, to find a good balance between the two parts of the system. In general, addressing these questions is as intricate as solving the theory reasoning problem itself, and even good heuristics cannot completely overcome these challenges.

One way to mitigate this issue is to adopt incremental techniques for background reasoning [37], as equality reasoning in ${}_3T^AP$ [35] or natural language processing in E-KRHyper [195]. It implies the use of algorithms that allow recording the results of the background reasoners computations and reusing this information for a later call. Then, the background reasoner can be called more often without the risk of processing useless computations. Moreover, an incremental background reasoner can reuse previously computed data multiple times, especially when multiple variations of a problem have to be solved.

Since incremental reasoning allows managing redundancy in the same branch, by avoiding redoing work, it can also be used to avoid redundancy between *multiple* branches. Indeed, by sharing information between the different calls to the background reasoner, previously computed solutions can be reused. For example, two branches sharing common terms could have the same closure and do not require any additional computation time. Moreover, keeping track of previous work can also avoid useless triggering of the background reasoner, for example, by calling it only when relevant axioms have been generated since the last call.

Finally, even if the cost to maintain such a structure is higher than a simple call to a background reasoner, the gain of reusing precomputed information in several branches offsets the overall cost. This incremental method, although suitable for tableaux, is also available for resolution-based method [24].

Typed Theories

In the last decade, we have seen that the multiplicity and diversification of problems have created the need to improve provers in order to be able to reason with theories. This is also the case when we aim to reason in the presence of *typed logics*, in which all the terms are *typed*. These logics are ubiquitous in industrial use and aim to provide a more intuitive representation of the world.

Most of the state-of-the-art automated provers support untyped or monomorphic logics, whereas specification languages or proof assistants are typically based on polymorphic formalisms. In order to be able to deal with typed problems, two approaches have emerged: erasing the types from the problem or, conversely, natively

managing types into a prover. The first one relies on an encoding of typed problems into untyped first-order logic [47, 87], which makes them accessible to a wider range of provers. For instance, the intermediate verification language Boogie 2 [165], which features polymorphic maps and higher-rank polymorphism, and Why3 [49], based on rank-1 polymorphism, which defines translations to a monomorphic logic [50, 91].

Conversely, the second option incorporates types directly into the terms themselves, adding additional rules to guarantee the *well-typedness* of the formulas. Such a system can be either monomorphic or polymorphic and is implemented in first-order logic in provers such as Zenon Modulo [73, 78], Alt-Ergo [89], Zipperposition [93] and ArchSAT [75], or the higher-order theorem prover LeoIII [223].

Chapter 3

Fairness Management in Tableau Proof-Search Procedures: a Concurrent Approach

Contents

The method of analytic tableaux for first-order logic presented in Section 1.2 has two main characteristics: a tree structure and the use of free variables. The former allows us to easily produce a proof, whereas the latter have shown their efficiency in the context of a *proof-search procedure*. However, the construction of a tableau relies on some key points in the procedure, in which decisions have to be made [34]. These choices have an impact on different aspects of the proof, ranging from selecting the substitution that closes a branch to determining the next branch to explore. Due to these multiple possibilities at each proof step, providing a fair, complete and efficient proof search can become challenging.

Parallel exploration of the branches can overcome these issues. The idea is to expand branches freely before trying to reconcile them and to find a common solution. Given the tree structure produced during the proof search, the method of analytic tableaux naturally fits with concurrent approaches. Moreover, concurrent computing offers a way to implement a proof-search procedure that explores branches simultaneously. Chapter 2 already gives an overview of concurrent proof-search procedures. However, they often focus primarily on parallel execution and performance and do not address directly the challenges inherent to tableaux. In contrast, we use concurrency not only as a way to take advantage of multi-core architectures but also as an algorithmic device that is useful even for sequential execution.

This chapter is divided into two parts: a description of existing fairness issues and a presentation of a concurrent solution. Section 3.1 offers a comprehensive exploration of the fairness concerns that arise during the construction of a complete proof-search procedure for free-variable tableaux and delves into the state-of-the-art sequential solutions designed to address these problems. These challenges are rooted in the intrinsic *destructive* nature of the closure rule, which in turn, by replacing free variables in the whole tableau upon a branch's closure, affects the unification possibilities of the remaining branches. By carefully examining the scenarios where unfairness arises, Section 3.2 introduces a concurrent proof-search procedure for first-order analytic tableaux that overcomes these issues.

3.1 Fairness Management in Tableau-Based Theorem Prover

The fairness challenges encountered in first-order tableaux stem from the existence of multiple choices at each stage of the proof-search process. Four main sources of unfairness have been introduced by [34], which can result in an unfair and consequently incomplete proof-search procedure.

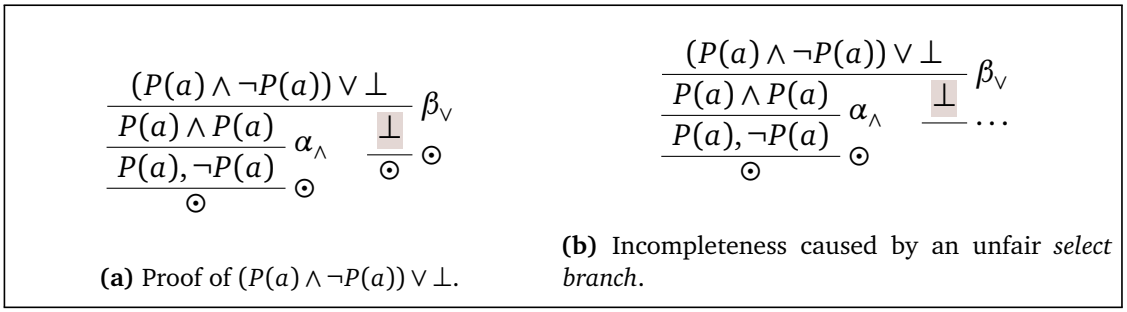
3.1.1 Incompleteness Induced by Fairness Issues

Typically, a considerable number of rules can be applied to any given free-variable tableau in the first-order logic context. To be more specific, the process involves selecting a branch B where a rule is to be applied, followed by the decision of whether an expansion rule or a closure rule should be employed. Previous studies in [34, 141] have explored the issues of unfairness in the proof-search procedure, identifying four crucial decision points:

1. The selection of a branch B (*select branch*).
2. Determining whether B should be closed or expanded (*select mode*).
3. If B is to be closed, the choice of a pair of complementary literals and thus a closing substitution (*select pair*).
4. If B is to be expanded, the selection of a formula to which an expansion rule is applied (*select formula*).

In the propositional case, achieving a strongly complete tableau proof-search procedure is straightforward, due to the finite number of choices available. To close any (valid) tableau, it suffices to select each non-atomic formula exactly once on each branch, in any order.

However, in the first-order case, the situation is more complex. Indeed, some formulas can be applied more than once, and multiple instances of the same formulas may be needed to close a tableau, making first-order logic undecidable. Proceeding to an arbitrary choice for a computation rule generally leads to an incomplete proof



(a) Proof of $(P(a) \wedge \neg P(a)) \vee \bot$.

(b) Incompleteness caused by an unfair *select branch*.

Figure 3.1: Proof and proof search illustrating the *select branch* problem.

procedure. Hence, attaining completeness in the first-order case requires careful consideration and strategies that go beyond simple arbitrary choices.

The phenomena of incompleteness arising from unfair selection strategies for these key decision points are illustrated through examples. Needless to say, these phenomena can also interact in intricate ways.

Select Branch

The first case of unfairness leading to incompleteness comes from the tree structure itself, i.e., the branches spawn. This incompleteness case is called *select branch* and happens when a branch is never expanded by the proof-search procedure.

Figure 3.1a and Figure 3.1b illustrate this case. In the proof (Figure 3.1a), both branches are explored, leading to a closed tree. However, an unfair proof-search procedure can fail to compute on of the branches (the right one in example Figure 3.1b), resulting in an unfinished proof.

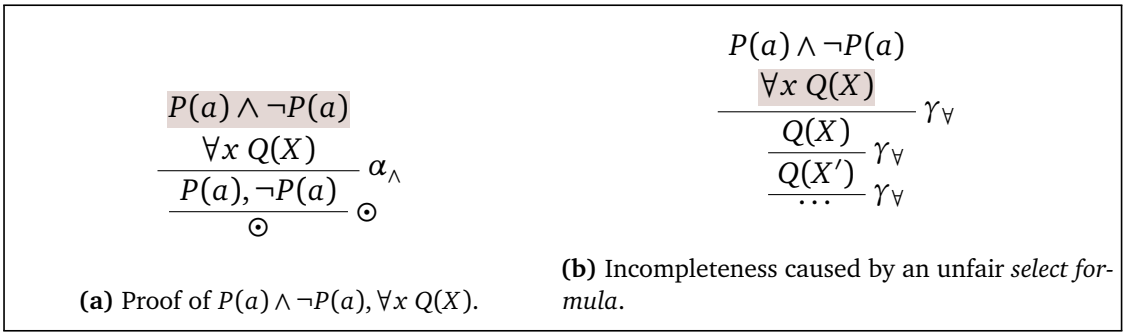
Nevertheless, to address this unfairness case, a *fair* proof-search procedure only needs to ensure that it does not terminate arbitrarily without proper justification, as all branches must be closed.

Select Formula

In order to expand the tableau, a formula is selected at each step of the proof search. Moreover, γ-formulas have the particularity of being able to be applied multiple times, i.e., to be put back into the set of available formulas. Due to these application, incompleteness can arise when a formula F is never selected during the proof search, i.e., when a set of formulas Γ is looped upon such that $F \notin \Gamma$.

This unfairness over formula computation can lead to completeness issues, for example, if F is involved in a closure rule, as shown in Figure 3.2a and Figure 3.2b. In this example, $P(a) \wedge \neg P(a)$ is needed to close the tree but the formula is never chosen, as $\forall x\ Q(x)$ is preferred during the proof search, delaying the expansion of the former indefinitely.

Since a formula is considered as processed once computed, it is relatively straightforward to ensure the proper treatment of α-, δ- and β- formulas, and thus overcome such incompleteness issue. However, specific attention must be given to γ-formulas, to prevent looping on them.



(a) Proof of $P(a) \wedge \neg P(a), \forall x\, Q(X)$.

(b) Incompleteness caused by an unfair *select formula*.

Figure 3.2: Proof and proof search illustrating the *select formula* problem.

Select Pair

A closure happens when two complementary literals, i.e., P and $\neg Q$, have been unified by a substitution σ such that $\sigma(P) = \sigma(Q)$. P and Q are thus called a *pair of complementary literal*. Moreover, the *destructive* nature of free-variable tableaux implies that the closure rule effectively replaces the free variables during the proof search. Therefore, the *select pair* problem can occur when the same closing pair of literals is consistently chosen, forbidding other pairs to be tried. This can lead to conflicts as different possibilities to close a branch may be mutually exclusive. Making the "wrong" choice, i.e., choosing a substitution which does not allow a *global* closure, and applying the incorrect substitution to the tableau can make it impossible to immediately perform the next branch closure, which could be more helpful.

This incompleteness case is illustrated in Figure 3.3a and Figure 3.3b. In this example, exploring the left branch produces a substitution that prevents direct closure of the right branch. Reintroducing the original quantified formula with a different free variable is not sufficient to close the right branch, because an applicable (standard) δ-rule creates a new Skolem symbol that will result in a different but equally problematic substitution every time a left branch is explored. Thus, systematically exploring the left branch before the right leads to choosing a closure between $P(X)$ and $\neg P(b)$, and thus to a non-termination of the search. Conversely, exploring the right branch first produces a substitution (which instantiates the free variable X with a rather than b) that closes both branches.

To handle this case, it may be necessary to repeat the sequence of expansion rule applications that led to the situation where the wrong choice was made. Additionally, proof-search steps may also need to be replayed on other branches, possibly impacted by the previous unfavorable choice. [1]

Select Mode

Deciding whether or not a branch should be closed is a common problem when designing a proof-search procedure. Typically, two approaches are considered: searching for a *global* closure that closes all branches simultaneously [123, 136], or allowing

[1] Note that this specific example can also be avoided by using a more advanced Skolemization, as presented in Section 2.1 and Chapter 7.

$$\dfrac{\dfrac{\dfrac{\dfrac{P(a)}{\neg P(b)}\quad\dfrac{\dfrac{\forall x.\, P(x) \Leftrightarrow \forall y\ P(y)}{P(X) \Leftrightarrow (\forall y\ P(y))}\,\gamma_\forall}{\dfrac{P(X), \forall y\ P(y)}{P(Y)}\,\gamma_\forall}\,\beta_\Leftrightarrow}{\sigma = \{X \mapsto a, Y \mapsto b\}}\,\odot_\sigma \qquad \dfrac{\neg P(X), \neg(\forall y\ P(y))}{\sigma = \{X \mapsto a\}}\,\odot_\sigma}$$

(a) Proof of $P(a), \neg P(b), \forall x.\, P(x) \Leftrightarrow \forall y\ P(y)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(a)}{\neg P(b)}\quad\dfrac{\dfrac{\forall x.\, P(x) \Leftrightarrow \forall y\ P(y)}{P(X) \Leftrightarrow \forall y\ P(y)}\,\gamma_\forall}{\dfrac{P(X), \forall y.\, P(y)}{\sigma = \{X \mapsto b\}}\,\odot_\sigma}\,\beta_\Leftrightarrow \qquad \dfrac{\dfrac{\neg P(X), \neg\forall y\ P(y)}{\neg P(f(X))}\,\delta_{\neg\forall}}{P(X') \Leftrightarrow \forall y\ P(y)}\,\gamma_\forall}{\dfrac{P(X'), \forall y\ P(y)}{\sigma = \{X' \mapsto b\}}\,\odot_\sigma}\,\beta_\Leftrightarrow \qquad \dfrac{\dfrac{\neg P(X'), \neg\forall y\ P(y)}{\neg P(f'(X_2))}\,\delta_{\neg\forall}}{\cdots}\,\gamma_\forall}{\sigma' = \{X' \mapsto f(X)\}}$$

(b) Incompleteness caused by an unfair *select pair*.

Figure 3.3: Proof and proof search illustrating the *select pair* problem.

a *local* closure that finds a substitution for only a specific subtree.

The *select mode* incompleteness case appears when branches are closed locally as soon as possible, preventing a *global* closure. As illustrated in Figure 3.4a, Figure 3.4b and Figure 3.4c, x is substituted by a or b whereas the desired substitution (i.e., the one able to close the whole tree) is c. However, this substitution can only be reached if $\partial R(X)$ was applied. Independently of which branch is closed first, the variable x gets "used up" by a substitution that blocks the closure of the other branch. Although it is possible to create a second instance of the γ-formula with a free variable, the same issue arises at the next level and continues repetitively. This example highlights one of the main problems of proof search in destructive free-variable tableaux.

This case of incompleteness is intrinsically linked to the *eager* closure, i.e., the strategy consisting of closing a branch as early as possible. Delaying the closure rule allows to overcome these issues, since this scenario cannot happen, but is also not usable in practice. A good balance between closure and γ-rule application is thus necessary to ensure the efficiency of the proof search.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg P(a)}{\neg Q(b)}}{\neg R(c)}}{\forall x.\,(P(x)\vee Q(x))\wedge \partial R(x).}\,\gamma_\forall}{(P(X)\vee Q(X))\wedge \partial R(X)}\,\alpha_\wedge}{P(X)\vee Q(X), \partial R(X)}\,\beta_\vee}{P(X)}\,\partial}{R(X)}\,\odot}{\{X\mapsto c\}} \qquad \dfrac{\dfrac{\dfrac{Q(X)}{R(X)}\,\partial}{\{X\mapsto c\}}\,\odot}$$

(a) Proof of $\neg P(a), \neg Q(b), \neg R(c), \forall x.\,(P(x)\vee Q(x))\wedge \partial R(x)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg P(a)}{\neg Q(b)}}{\neg R(c)}}{\forall x.\,(P(x)\vee Q(x))\wedge \partial R(x).}\,\gamma_\forall}{(P(X)\vee Q(X))\wedge \partial R(X)}\,\alpha_\wedge}{P(X)\vee Q(X), \partial R(X)}\,\beta_\vee}{P(X)}}{\{X\mapsto a\}}\cdots \qquad \dfrac{Q(X)}{\{X\mapsto b\}}\cdots$$

(b) Incompleteness caused by an unfair *select mode* — step 1.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg P(a)}{\neg Q(b)}}{\neg R(c)}}{\forall x.\,(P(x)\vee Q(x))\wedge \partial R(x).}\,\gamma_\forall}{(P(a)\vee Q(a))\wedge \partial R(a)}\,\alpha_\wedge}{(P(a)\vee Q(a)), \partial R(a)}\,\beta_\vee}{P(a)}}{\{X\mapsto a\}}\,\odot \qquad \dfrac{\dfrac{Q(a)}{R(a)}\,\partial}{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(P(X')\vee Q(X'))\wedge \partial R(X')}{(P(X')\vee Q(X')), \partial R(X')}\,\alpha_\wedge}{P(X')}\,\beta_\vee}{\{X'\mapsto a\}}\cdots \qquad \dfrac{Q(X')}{\{X'\mapsto b\}}\cdots}}$$

(c) Incompleteness caused by an unfair *select mode* — step 2.

Figure 3.4: Proof and proof search illustrating the *select mode* problem.

3.1.2 Sequential Approaches and Existing Solutions

Dealing with fairness issues mentioned in Section 3.1.1 is possible in a sequential way. This section focuses on sequential provers and describes the mechanisms commonly used to manage them: the closure style and the bounded search. Additionally, [34] presents some improvements in this regard.

Closure Although the only way to close a tableau is to find a *global* substitution that closes all the branches, finding such a substitution at once is highly inefficient. For example, a naive approach attempting to find a *global* substitution at each step would certainly be a complete approach, but cannot work in practice [124].

Instead, trying to close branches on the fly and propagate the information about the substitutions is a more common approach, even if it is prone to completeness issues. [136] proposes an incremental closure rule, which keeps track of substitutions for each branch and eventually closes the whole tableau at once. This technique is implemented in the prover Princess [211]. By adding the required instances to a branch instead of instantiating the free variables, Zenon [65] avoids the destructive side of free-variable tableaux, and thus the associated resulting completeness issues. It also implements a pruning mechanism to avoid duplicated work for branches that share variables.

Breath-First Search and Iterative Deepening In order to design an efficient proof-search procedure for a formula F, all the generative proof trees for F have to be explored in a reasonable way. Unrestricted depth-first search is excluded because of the fairness issues discussed above, i.e., the perpetual application of a rule on a γ-formulas as presented on the select *formula* case or the never-chosen branch of the select *branch* one. Thus it remains the breadth-first search (BFS) and the iterative deepening depth-first search (IDDFS).

A fundamental advantage of IDDFS over BFS is that it can be implemented efficiently using a γ-rule application limit and backtracking as in [38]. Although this leads to acceptable performance of tableau-based automated theorem provers [162], the asymptotic complexity of IDDFS is no worse than that of BFS, it should be stressed that an IDDFS search is only a compromise as a complete selection strategy for destructive free-variable tableaux without backtracking with local closure is not yet available [34].

Since the *select branch* and *select formula* cases can be easily managed, in the following we focus on the problem of an unfair *select mode* and *select pair* problems and see how concurrency can address these fairness issues.

3.2 The Use of Concurrency for an Efficient Fairness Management

This section describes a concurrent proof-search procedure which, by design, deals with the two main sources of unfairness (and therefore incompleteness): the *select*

pair and the *select branch* problems. The key idea is to explore branches at the same time, allowing simultaneous closure (possibly with incompatible substitutions). Then, a reconciliation phase happens, which must lead to finding an agreement between the substitutions returned by the different branches. However, the use of concurrency also encountered challenges, such as the dependencies between siblings caused by shared free variables, that can be tackled with multiple strategies such as backtracking and forbidden substitutions.

3.2.1 State of the branches and Closure Management

First of all, we define the state of a branch used in the proof-search procedure, as it is strongly connected to the concrete algorithms developed in the following section. A branch state is conditioned by its closing status, i.e., not closed or closed with a *local* or *non-local* substitution. Recall that, contrary to a *non-local* substitution, a *local* one is necessarily compatible with the rest of the tree. Each step of the proof search can be mapped to an abstract proof state denoted (T, σ), where T is the tree formed by the union of the branches manipulated by the different processes and σ the composition of all the substitutions used to close $\odot$-branches. The $\odot$ state is part of three different statuses that each node is additionally annotated with.

Definition 3.1: Closed Node

> $\odot$ indicates that the node is *locally closed*: the subtree rooted in this node has been closed with a local substitution.

Definition 3.2: Stalled Node

> $\bowtie$ indicates that the node is *stalled*: a *non-local* substitution has been found for the subtree rooted in this node.

Definition 3.3: Active Node

> ✠ indicates that the node is *active*: the subtree rooted in this node is still being expanded by the proof-search procedure, or that no more expansion rules are applicable.

With these definitions in place, the main procedures and the transitions between these branch states can be described, detailing how the proof-search process operates and how branches switch between different states.

3.2.2 Tableau Representation and Abstract Procedure Rules

This section implements the procedure rules seen in Section 1.2. Let (T, σ) and (T', σ') be two tableaux such that $(T, \sigma) \hookrightarrow_R (T', \sigma')$ by using a rule R. Let Γ be the set of sets of formulas representing T. Let B be the branch (i.e., a set of formulas) and f the formula of B on which the rule R has to be applied to generate (T', σ'). We denote $\Psi = \Gamma \setminus B$ the unprocessed set of sets of formulas on the other branches.

We describe an abstraction of the rule to represent the application of a rule in a tableau proof-search procedure. This application is symbolized as a transition between two states of a proof. A state of a proof is a quintuplet $\langle \Gamma, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, \Sigma, \chi \rangle$ with:

- Γ is the set of sets of formulas representing the tree.
- $\Theta_{\text{backtrack}}$ a set of substitutions that may be applied subsequently.
- $\Theta_{\text{forbidden}}$ a set of substitutions that may be forbidden subsequently.
- Σ the function which takes a branch and returns the substitution currently applied to the branch.
- χ the set of substitutions currently forbidden in T.

Informally, a state represents a tableau (T, σ) plus a set of substitutions $\Theta_{\text{backtrack}}$, which haven't been tested yet, and as set $\Theta_{\text{forbidden}}$ of substitutions which have to be possibly forbidden in a subsequent step. χ is a set of substitutions which are currently forbidden. It interferes directly with the closure rule.

We describe a rule system to explain the application of a rule in our procedure. We implement the rule defined in Section 1.2, as well as two new rules to describe the backtracking mechanism. Thus, considering a rule R, the transition between two states is as follows:

- R is an α-rule and $f \rightsquigarrow_R f'$:
  $\langle \Psi \cup \{B \cup \{f\}\}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, \Sigma, \chi \rangle \rightarrow_\alpha$
  $\langle \Psi \cup \{B \cup \{f'\}\}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, \Sigma, \chi \rangle$
- R is a β-rule and $f \rightsquigarrow_R f'_1, \ldots, f'_n$:
  $\langle \Psi \cup \{B \cup \{f\}\}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, \Sigma, \chi \rangle \rightarrow_\beta$
  $\langle \Psi \cup \bigcup_{i=1}^{n} \{B \cup \{f'_i\}\}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, \Sigma, \chi \rangle$
- R is a δ-rule and $f \rightsquigarrow_R f'_{[x \mapsto sko(\text{FV})]}$:
  $\langle \Psi \cup \{B \cup \{f\}\}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, \Sigma, \chi \rangle \rightarrow_\delta$
  $\langle \Psi \cup \left\{B \cup \left\{f'_{[x \mapsto sko(\text{FV})]}\right\}\right\}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, \Sigma, \chi \rangle$
- R is a γ-rule and $f \rightsquigarrow_R f'_{[x \mapsto X]}$:
  $\langle \Psi \cup \{B \cup \{f\}\}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, \Sigma, \chi \rangle \rightarrow_\gamma$
  $\langle \Psi \cup \left\{B \cup \left\{f, f'_{[x \mapsto X]}\right\}\right\}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, \Sigma, \chi \rangle$

- R is a non-local closure rule. It generates a set of substitutions σ which close a branch B of T (w.r.t χ). These substitutions are added to $\Theta_{\text{backtrack}}$ to be tried later as a global solution:
  $\langle \Psi \cup \{B\}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, \Sigma, \chi \rangle \rightarrow_{\odot_{non-local}}$
  $\langle \Psi \cup \{B\}, \Theta_{\text{backtrack}} \cup \sigma, \Theta_{\text{forbidden}}, \Sigma, \chi \rangle$

- R is a global closure rule. It can be triggered when there exists a σ such that σ is applied to all the branches and all the branches are closed. This rule closes the whole tree.
  $\langle \Gamma, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, \forall B.\ \Sigma(B) = \sigma, \chi \rangle \rightarrow_{\odot_{global}} \textit{Success}$

- R is a choice rule. It chooses a substitution σ among $\Theta_{\text{backtrack}}$ and applies it to all the branches. It can lead to a common agreement or to choose another substitution. Thus, σ is removed from $\Theta_{\text{backtrack}}$ and added to $\Theta_{\text{forbidden}}$. This rule can be triggered when $\Theta_{\text{backtrack}}$ is not empty, after that all the branches have found a substitution, or when no other rule is applicable:
  $\langle \Gamma, \Theta_{\text{backtrack}} \cup \{\sigma\}, \Theta_{\text{forbidden}}, \Sigma, \chi \rangle \rightarrow_{Choice}$
  $\langle \Gamma, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}} \cup \{\sigma\}, \forall B \in \Gamma.\ \Sigma(B) = \sigma, \chi \rangle$

- R is a forbidden rule. It can be triggered when no other rule can be applied and there exists at least one branch B such that $\Sigma(B) = \emptyset$. This situation happens when a branch cannot be closed with the current substitution. The rule can be triggered if there is no backtracking option (i.e., $\Theta_{\text{backtrack}} = \emptyset$) and if $\Theta_{\text{forbidden}}$ is not empty. It takes all the substitutions stored in $\Theta_{\text{forbidden}}$ and transfers them into χ to resume the proof search:
  $\langle \Gamma, \emptyset, \Theta_{\text{forbidden}}, \Sigma, \chi \rangle \rightarrow_{Forbidden}$
  $\langle \Gamma, \emptyset, \emptyset, \forall B \in \Gamma.\ \Sigma(B) = \emptyset, \Theta_{\text{forbidden}} \rangle$

- R is a global non-closure rule. It can be triggered when there is no backtracking nor forbidden option, and where no extension rule is available. This rule leads to a failure to find a solution.
  $\langle \Gamma, \emptyset, \emptyset, \Sigma, \chi \rangle \rightarrow_{O_{global}} \textit{Failure}$

Intuitively, these rules try to find a solution for each branch individually and then start an agreement process. The agreement is global, meaning that the solution of every branch is tried on the whole tree. However, this procedure leads to a lot of useless attempts, because the chosen substitutions are tried without any restriction. Thus, we adapt these rules to work in a more local way, i.e., search to find an agreement with their neighbors before broadcasting it in a larger way. To do so, some changes are needed:

- The quintuplet is extended with an additional function ζ, which takes a branch and returns the solutions found by the branch. The behavior of the non-local closure rule is thus modified: the found substitution is no longer added to $\Theta_{\text{backtrack}}$, but stored into ζ instead.

- $\Theta_{\text{backtrack}}$ and $\Theta_{\text{forbidden}}$ are now functions, each one dedicated to a branch, just as Σ and ζ.

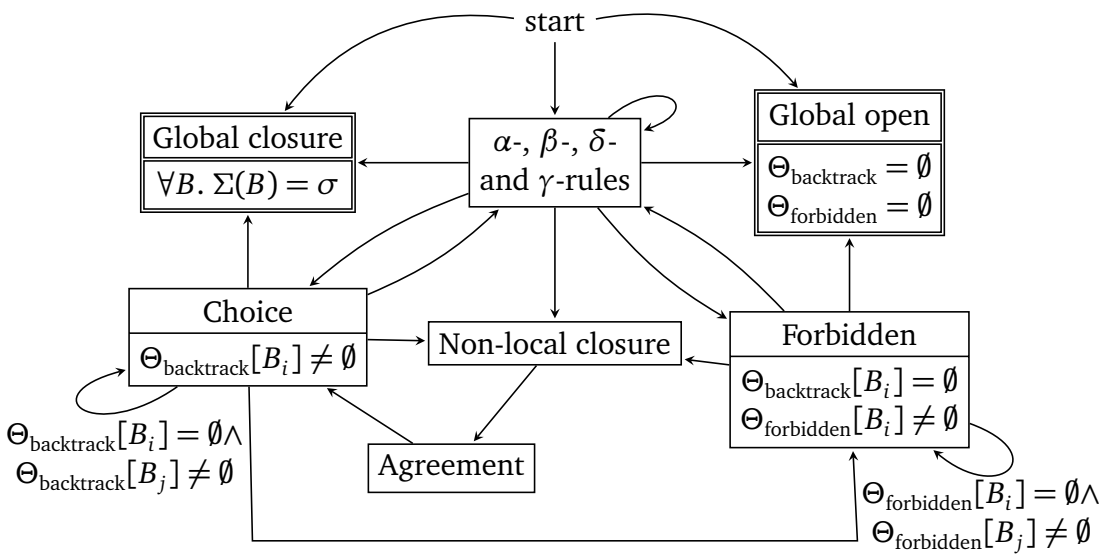


Figure 3.5: Interaction and application conditions of the abstract procedure rules.

- A new rule needs to be defined: the agreement rule. It can be triggered when all the branches of a subset $B_1, \ldots, B_n$ of T have all reached a non-local closure. Thus, it takes all the solutions found in the corresponding ζ and performs a local choice, in the same ways as before:
  $\langle \Psi \cup \{B_1, \ldots, B_n\}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, \Sigma, \chi, \zeta \rangle \rightarrow_{\mathit{Agreement}}$
  $\langle \Psi \cup \{B_1, \ldots, B_n\}, \Theta_{\text{backtrack}}(B_i) = \zeta(B_i)\ (1 \leq i \leq n), \Theta_{\text{forbidden}}, \Sigma, \chi, \emptyset \rangle$

- The choice rule can now be triggered locally after the application of the agreement rule, no more a non-local closure one. Moreover, it can only be applied if there is at least one backtracking option for the related subset of formulas.

- The forbidden rule also changes to be able to deal with a subset of branches. It can be triggered when there is no backtracking option for this subset (and forbidden option for it):
  $\langle \Psi \cup \{B_1, \ldots, B_n\}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, \Sigma, \chi, \zeta \rangle \rightarrow_{\mathit{Forbidden}}$
  $\langle \Psi \cup \{B_1, \ldots, B_n\}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}(B_{i_{(1 \leq i \leq n)}}) = \emptyset,$
  $\Sigma(B_{i_{(1 \leq i \leq n)}}) = \emptyset, \chi(B_i) = \Theta_{\text{forbidden}}(B_i)\ (1 \leq i \leq n), \zeta \rangle$

Interactions between these rules are presented in Figure 3.5. The lower part of a node represents the application condition of the rule, with potentially additional restrictions on the edges, and B_i and B_j two sets of branches. These rules are used as a base to implement an imperative proof-search procedure, presented in the next section.

3.2.3 A Concurrent Proof-Search Procedure

This section presents the concurrent proof-search procedure. The procedures are described using the concurrent semantics defined in Section 1.3.3. The proof search is carried out concurrently by processes corresponding to branches of the tableau and relies solely on parent-children communications. Processes are started upon the

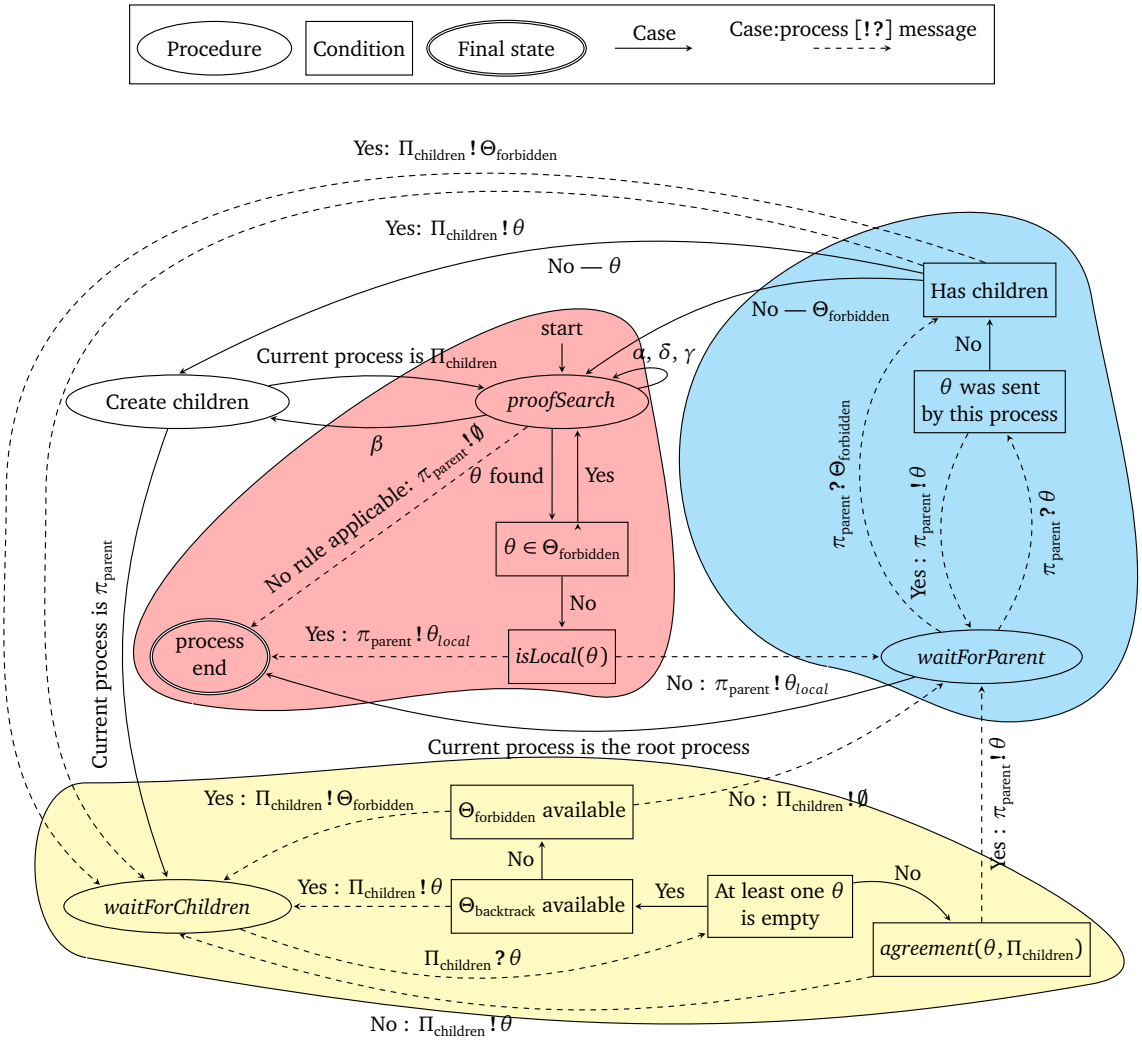


Figure 3.6: The proof-search procedure executed by individual processes

application of a β-rule, one for each new branch. Communications between processes take two forms: a process may send a set of closing substitutions to its parent, or a parent may send a pair of substitutions to its children. The first element of this pair is a substitution that closes at least one of the children and needs to be tried by its siblings. The second element is a set of forbidden substitutions, i.e., substitutions that were tried by a parent node but cannot close a sibling branch. Such unsuccessful substitutions are prohibited for the current node and its descendants for the rest of their proof search. The proof search is performed by the *proofSearch* procedure (described in Procedure 3), which calls the *waitForParent* and *waitForChildren* procedures (available in Procedures 4 and 5, respectively). Their interactions are illustrated in Figure 3.6.

Agreement Mechanism The core of the following procedures is the agreement mechanism. Coarsely, all the children of a node n search for their own solution, and then n applies multiple agreement phases to find a solution that satisfies all the children. This is obtained by choosing an arbitrary substitution among the answers,

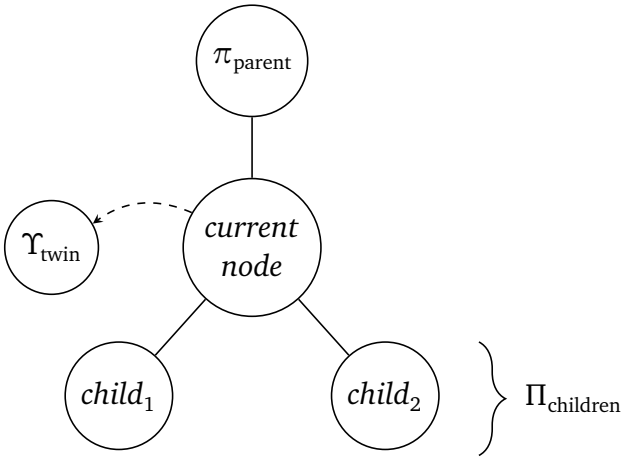


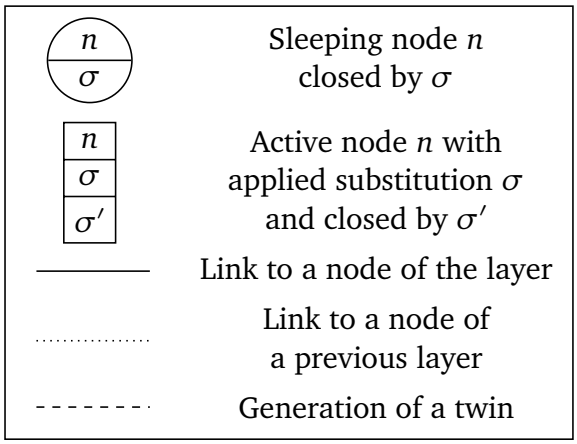
Figure 3.7: Node structure and local vision extended to twin processes.

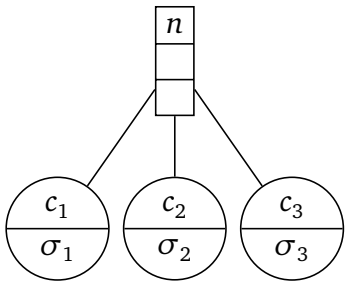
and by sending it to the other children. The final substitution is thus built from one step to the next, by adding constraints coming from each child. This behavior relies on a specific type of children, called *twin* and denoted Υ_{twin}, as well as the notion of *layers* of agreement.

A twin represents an "alternative version" of the node. This twin can in turn build a sequence of twins, each one being an evolution of the previous one. A node can only have two twins: the previous one and the next one in the sequence. The local vision of a node enhanced by twins processes is available in Figure 3.7.

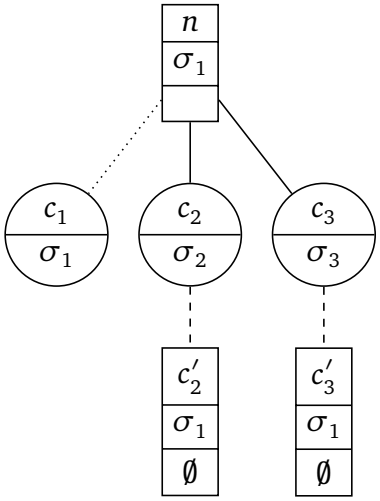
A layer is characterized by a pair $(\sigma, c_{i_{(1\leq i\leq n)}})$ where c_i are active *twins* (or children for the initial layer) on which the substitution σ is applied. To give an insight, the first layer contains all the children of the branch, the second one only those that have not found the same substitution as the one selected during the first step, and so on. The children that have already agreed are considered as *sleeping*, whereas those that keep working to find a complementary substitution are denoted as *active*. At each new layer, a *twin* of each active node is created, constrained by the substitution selected at the previous layer. In case no agreement is reached, the children of the current layer are killed, and the agreement mechanism restarts from the previous layer, with another potential candidate. Some substitutions can also be forbidden for a given layer, forcing the related nodes to submit other propositions. For this purpose, Π_{children} evolves to become a map of sets of processes, in which the index corresponds to the agreement layer and the corresponding set to the active children of the layer. For instance, $\Pi_{\text{children}}[0]$ corresponds to the active children of the initial layer.

An illustration of the agreement layer process is available in Figure 3.8. In this example, three substitutions σ_1, σ_2 and σ_3 are respectively found by c_1, c_2 and c_3 during the initial agreement phase, i.e., the first layer (Figure 3.8a). Then, σ_1 is chosen, putting c_1 to sleep and creating two twins: c'_2 and c'_3. This pair $(\sigma_1, \{c'_2, c'_3\})$ represents a second layer (by opposition the the first layer with no substitution and three children). Unfortunately, the children cannot be closed if σ_1 is applied, and both of them answer an empty set, symbolizing the lack of substitution (Figure 3.8b). Then, a new layer (on the second level too) is created, with a new candidate substitution σ_2 and two twins: c'_1 and c'_3 (Figure 3.8c). From this layer emerged two new solutions, complementary

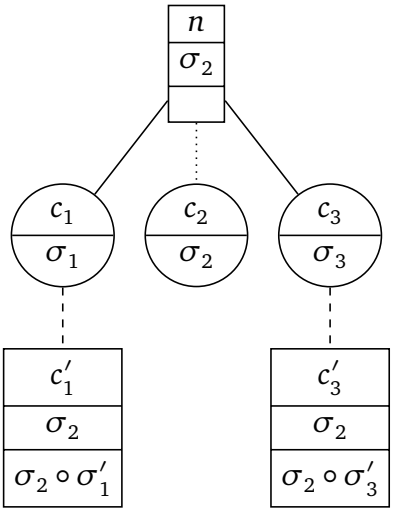




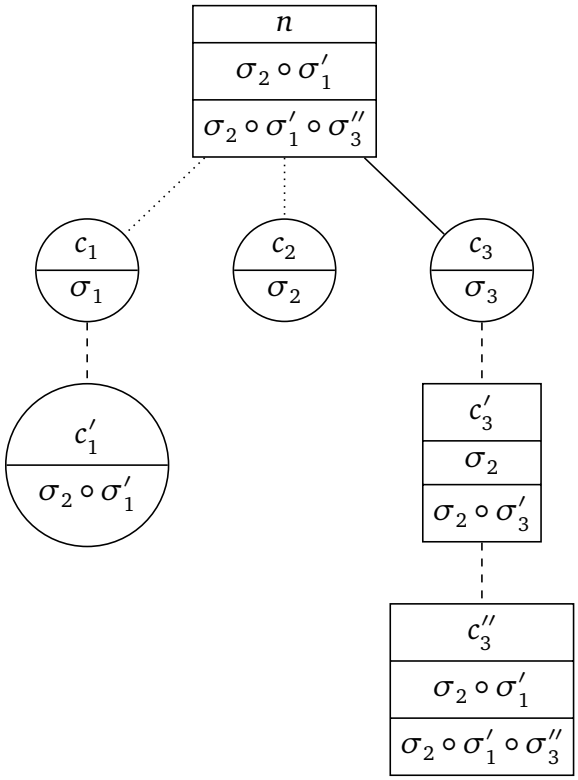
(a) First layer of agreement — solution found by the children.



(b) Second Layer of agreement — instantiation by σ_1 — no solution found.



(c) Second layer of agreement — instantiation by σ_2 — compatibles solutions found by the active children.



(d) Second layer of agreement — instantiation by $\sigma_2 \circ \sigma'_1$ — agreement found among all the children.

Figure 3.8: Agreement layer mechanism.

to σ_2: $\sigma_2 \circ \sigma'_1$ and $\sigma_2 \circ \sigma'_3$. Then, an even more deeper agreement layer is created, by selecting $\sigma_2 \circ \sigma'_1$ and the twin c''_3. The node finds a substitution $\sigma_2 \circ \sigma'_1 \circ \sigma''_3$, that brings agreement with all the children of the layer (itself only in this case) (Figure 3.8d). The resulting substitution is a refinement of those selected within the previous layers, the sleeping node necessarily agrees, and thus the whole subtree is closed.

Related Functions The procedures call some auxiliary functions for which the detailed implementation is relatively straightforward. Then, instead of explaining them in the detail, we solely give an overview of their behavior.

First of all, four functions are related to the rules and their applications:

- *nextApplicableRule*: this function computes the next applicable rule among the α-,δ-,β- and γ-rules, following this priority order, as well as the formula on which the rule has to be applied.
- *applyRule*: this function takes a rule and a branch of a tableau and applies the given rule to the branch.
- *applyClosingRule*: this function takes a branch and searches for a closing rule, excepting those belonging to $\Theta_{\text{forbidden}}$.
- *updateLimit*: this function takes a branch B, a formula f and a γ-rule application limit and return the new limit after the application of the formula. If all the formula in B excepted f have been reintroduced n times, this function returns $n-1$, and n otherwise.

Then, two functions relate to the substitutions, either by giving information about them or by helping the main procedure itself:

- *isLocal*: this function returns true if the given substitution contains solely *local* variables, false otherwise.
- *choice*: this function chooses an arbitrary substitution among a set of substitutions.

Some functions are designed to give information about the current process and its links to others:

- *hasParent*: this function returns true if the current node has a parent process.
- *isSleeping*: this function returns true if the given process is sleeping, which is made possible thanks to *sleep* operation. It takes another function as a parameter, which represents the action to be performed after it wakes up.
- *isConstrained*: this function returns true if the current process is constrained by a substitution at this layer, i.e., is currently trying a substitution on its children, and false otherwise.

Finally, two functions allow us to interfere directly with the usual parent-child communication:

- *connect*: this function creates a communication channel between two nodes that are not parent and child. It is used to allow a parent node and a twin of a child to communicate.

- *update*: this function takes a layer index i, the set of children Π_{children}, the map of backtracking $\Theta_{\text{backtrack}}$ and forbidden $\Theta_{\text{forbidden}}$ substitutions and a substitution σ, and processes multiple operations. First of all, for the layer of agreement of index i, it kills all the open twins and wakes up all the children of the layer. Secondly, it adds σ to the forbidden substitutions of the layer $\Theta_{\text{forbidden}}[i]$, and removes it from the backtracking map $\Theta_{\text{backtrack}}[i]$.

In contrast with the three main procedures that can be seen as a state of a process (*proofSearch*, *waitForParent* and *waitForChildren*), the previous functions are strictly subcomponents of the procedures and contribute to advancing toward the next step in the proof search.

The Proof-Search Procedure The *proofSearch* procedure initiates the proof search for a branch. It first attempts to apply the closure rule, excluding closing substitutions that are subsumed by the ones in $\Theta_{\text{forbidden}}$ (Line 2). If one of the closing substitutions is local to the node corresponding to that process, it is reported, and the process terminates (Lines 4-6). If only non-local closing substitutions are found, they are reported and the process executes *waitForParent* (Lines 7-9). Otherwise, the procedure applies tableau expansion rules according to the priority: $\alpha \prec \delta \prec \beta \prec \gamma$. If a β-rule is applied, new processes are started, and each of them executes *proofSearch* on the newly created branch, while the current process executes *waitForChildren* (Lines 20-22). γ-rules are subject to a limit l to avoid unlimited application of a rule on a universal formula (Line 26). When no more rules can be applied, a process indicates the failure to close the tableau to its parent (Line 15). If the process has no parents, i.e., this is the root process, the proof search restarts with a higher limit (Line 17).

The Child's Procedure The *waitForParent* procedure is executed by a process P after it has found closing non-local substitutions. Since such substitutions may prevent closure in other branches, the parent will eventually send another candidate substitution, or prohibit a set of substitutions. *waitForParent* waits until such a substitution or restriction is received (Line 2). If a substitution σ' is received, the process controls the substitution. If σ' was previously sent by P itself, it means that the parent process is currently trying this substitution on the P's siblings. Thus, it sends σ' back, because it necessarily agrees with it (Line 6) and switches to a *sleeping* state, meaning it does not carry an active proof search and wait for another information from its parent (Line 7). Otherwise, the substitution is either sent to the child processes (Line 9), or applied to the current process itself. In order to keep the backtracking point unaltered, a new *twin* process is created, executing the proof search on $\sigma'(P)$. P is thus put into a sleeping state, waiting for its parent to wake it up (Lines 13-16). To allow the twin and the parent to exchange messages, a communication channel is created (Line 15). A twin process can only be created by a leaf, and its behavior is illustrated in Figure 3.9.

Procedure 3: *proofSearch*

Data: a branch B, a map $\Theta_{\text{forbidden}}$ of forbidden substitutions, a limit $l \in \mathbb{N}$ on the number of γ-rule applications.

```
1  begin
2      var Θ ← applyClosingRule(B, Θ_forbidden)
3      for θ ∈ Θ do
4          if isLocal(θ) then
5              π_parent ! θ
6              return
7      if Θ ≠ ∅ then
8          π_parent ! Θ
9          waitForParent(B, Θ, ∅, ∅, l, 0)
10     else
11         var rule, f ← nextApplicableRule(B, l)
12         switch rule do
13             case No rule applicable do
14                 if hasParent() then
15                     π_parent ! ∅
16                     waitForParent(B, ∅, ∅, ∅, 0, 0)
17                 else proofSearch(B, ∅, l + 1)
18             case α, δ do
19                 proofSearch(applyRule(rule, B), ∅, l)
20             case β do
21                 for B′ ∈ applyBetaRule(B) do
22                     start proofSearch(B′, ∅, l)
23                 waitForChildren(B, ∅, ∅, ∅, l, 0)
24             case γ do
25                 l′ ← updateLimit(B, f, l)
26                 proofSearch(applyRule(rule, B), ∅, l′)
```

The other case available is the one in which a set $\Theta_{\text{forbidden}}$ of prohibited substitutions has been received. Thus, P starts by updating $\Theta_{\text{backtrack}}$ and Θ_{sent} with this new information (Lines 18-19), to prevent the proof search for finding irrelevant substitutions on this layer. Recall that the node is currently on an agreement mechanism, meaning that it has already found potential closures. Thus, if the previous closures are compatible with the forbidden substitutions, they are returned (Line 21). Otherwise, either the prohibition is sent to the children of the current process (Line 24), either it resumes its proof search to find new closures (Line 27). It is important to note that in this case, the proof search resumes on the process itself, not in a twin, to increase the backtracking point. Thus, a node that has found a contradiction is

Procedure 4: *waitForParent*

Data: a branch B, a set Θ_{sent} of substitutions sent by this process to its parent, a map $\Theta_{\text{backtrack}}$ of set of candidate substitutions used for backtracking, a map $\Theta_{\text{forbidden}}$ of set of forbidden substitutions, a limit $l \in \mathbb{N}$ on the number of γ-rule applications, a level of layer i for the agreement mechanism.

1 **begin**
2 $\quad \pi_{\text{parent}}$ **?** $(\sigma', \Theta_{\text{forbidden}}{}')$
3 $\quad$ **switch** $\sigma', \Theta_{forbidden}{}'$ **do**
4 $\quad\quad$ **case** $\sigma' \neq \emptyset$ **do**
5 $\quad\quad\quad$ **if** $\sigma' \in \Theta_{sent}$ **then**
6 $\quad\quad\quad\quad \pi_{\text{parent}}$ **!** $\{\sigma'\}$
7 $\quad\quad\quad\quad sleep(waitForParent(B, \Theta_{\text{sent}}, \Theta_{\text{backtrack}}, \sigma, \Theta_{\text{forbidden}}, l, i))$
8 $\quad\quad\quad$ **else if** $\Pi_{children} \neq \emptyset$ **then**
9 $\quad\quad\quad\quad$ **for** $\pi \in \Pi_{children}[i]$ **do** π **!** $(\sigma', \emptyset)$
10 $\quad\quad\quad\quad \Theta_{\text{forbidden}}[i] \leftarrow \Theta_{\text{forbidden}}[i] \cup \{\sigma'\}$
11 $\quad\quad\quad\quad waitForChildren(B, \Theta_{\text{sent}}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, l, i)$
12 $\quad\quad\quad$ **else**
13 $\quad\quad\quad\quad B' = \sigma'(B)$
14 $\quad\quad\quad\quad$ **start** $proofSearch(B', \emptyset, l)$
15 $\quad\quad\quad\quad connect(\pi_{\text{parent}}, \Upsilon_{\text{twin}})$
16 $\quad\quad\quad\quad sleep(waitForParent(B, \Theta_{\text{sent}}, \emptyset, \emptyset, l, 0))$
17 $\quad\quad$ **case** $\Theta_{forbidden}{}' \neq \emptyset$ **do**
18 $\quad\quad\quad \Theta_{\text{backtrack}}[i] \leftarrow \Theta_{\text{backtrack}}[i] \setminus \Theta_{\text{forbidden}}{}'$
19 $\quad\quad\quad \Theta_{\text{sent}} \leftarrow \Theta_{\text{sent}} \setminus \Theta_{\text{forbidden}}{}'$
20 $\quad\quad\quad$ **if** $\Theta_{sent} \neq \emptyset$ **then**
21 $\quad\quad\quad\quad \pi_{\text{parent}}$ **!** Θ_{sent}
22 $\quad\quad\quad\quad waitForParent(B, \Theta_{\text{sent}}, \Theta_{\text{backtrack}}, \Theta_{\text{forbidden}}, l, i)$
23 $\quad\quad\quad$ **else if** $\Pi_{children}[i] \neq \emptyset$ **then**
24 $\quad\quad\quad\quad$ **for** $\pi \in \Pi_{children}[i]$ **do** π **!** $(\emptyset, \Theta_{\text{forbidden}}{}')$
25 $\quad\quad\quad\quad waitForChildren(B, \Theta_{\text{sent}}, \Theta_{\text{backtrack}}, \emptyset, \Theta_{\text{forbidden}}{}', l, i)$
26 $\quad\quad\quad$ **else**
27 $\quad\quad\quad\quad proofSearch(B, \Theta_{\text{forbidden}}, l)$

not locked into this step forever. The process may also be terminated by its parent (via the **kill** instruction) during the execution of this procedure if another child process from the same parent process cannot be closed, and no backtracking option is available. That is to say, the proof search is interrupted as soon as a branch cannot be closed, and either another substitution is tried or the process is killed and the scheme repeats until reaching the root.

Procedure 5: *waitForChildren*

Data: a branch B, a set Θ_{sent} of substitutions sent by this process to its parent, a map $\Theta_{\text{backtrack}}$ of set of candidate substitutions used for backtracking, a map $\Theta_{\text{forbidden}}$ of set of forbidden substitutions, a limit $l \in \mathbb{N}$ on the number of γ-rule applications, a level of layer i for the agreement mechanism.

```
1  begin
2      var subst ← f_⊥
3      while ∃ π ∈ Π_children[i]. subst[π] = ⊥ do
4          π ? subst[π]
5          if subst[π] = ∅ then
6              if ∃ θ ∈ Θ_backtrack[i] then
7                  update(Π_children, Θ_backtrack, Θ_forbidden, θ, i)
8                  for π ∈ Π_children[i] do π ! (θ, ∅)
9                  waitForChildren(B, Θ_sent, Θ_backtrack, Θ_forbidden, l, i)
10             else if isConstrained() then
11                 update(Π_children, Θ_backtrack, Θ_forbidden, ∅, i)
12                 for π ∈ Π_children[i] do π ! (∅, Θ_forbidden[i])
13                 waitForChildren(B, Θ_sent, Θ_backtrack, Θ_forbidden, l, i)
14             else if i > 0 then
15                 update(Π_children, Θ_backtrack, Θ_forbidden, ∅, i − 1)
16                 if ∃ θ ∈ Θ_backtrack[i − 1] then
17                     for π ∈ Π_children[i − 1] do π ! (θ, ∅)
18                     update(Π_children, Θ_backtrack, Θ_forbidden, θ, i − 1)
19                 else
20                     for π ∈ Π_children[i − 1] do π ! (∅, Θ_forbidden[i − 1])
21                 waitForChildren(B, Θ_sent, Θ_backtrack, Θ_forbidden, l, i − 1)
22             else
23                 if hasParent() then
24                     π_parent ! ∅
25                     waitForParent(B, Θ_sent, Θ_backtrack, Θ_forbidden, l, 0)
26                 else
27                     proofSearch(B, ∅, l + 1)
28         if isSleeping(π) then Π_children[i] ← Π_children[i] \ {π}
29     if ∃ θ ∈ Θ_backtrack . agreement(θ, Π_children[i]) then
30         π_parent ! {θ}
31         update(Π_children, Θ_backtrack, Θ_forbidden, θ, i)
32         waitForParent(B, Θ_sent ∪ {θ}, Θ_backtrack, Θ_forbidden, l, i)
33     else
34         σ′ ← choice(subst)
35         for π ∈ Π_children[i] do π ! (σ′, ∅)
36         Π_children[i + 1] ← Π_children[i]
37         Θ_backtrack[i + 1] ← subst \ {σ′}
38         Θ_forbidden[i + 1] ← Θ_forbidden[i] ∪ {σ′}
39         waitForChildren(B, Θ_sent, Θ_backtrack, Θ_forbidden, l, i + 1)
```

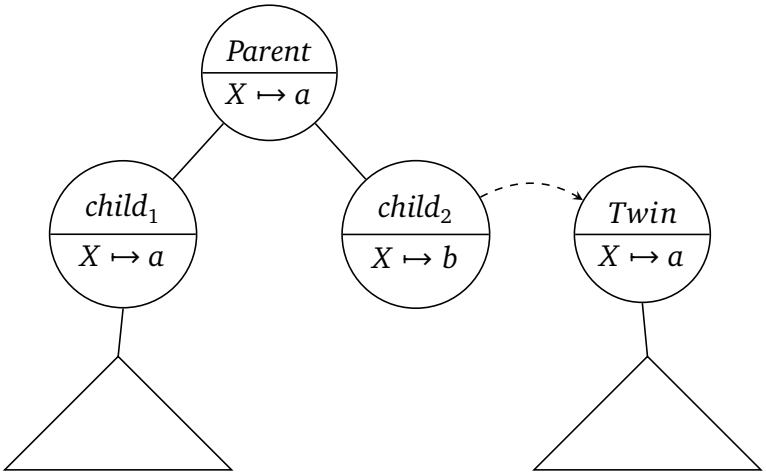


Figure 3.9: Twin behavior in the proof search.

The Parent's Procedure The *waitForChildren* procedure is executed by a process after the application of a β-rule and the creation of child processes. The set of substitutions sent by each child is stored in a map *subst* (Line 2), initially undefined everywhere ($f_\perp$). This case captures the agreement mechanism, i.e., it attempts to find at least one common substitution between all the children.

The first cases that are managed are those in which at least one of the children of the layer cannot close itself, which encompasses multiple situations. The first one occurs when a child is unable to close its own branch even without any constraints from its siblings, indicating a lack of solution. In this case, the parent process sends a failure message ($\emptyset$) to its parent (Lines 24-25) or resumes the proof search with a higher bound (Line 27).

Otherwise, three backtracking cases can happen. If there are substitutions at this layer that have not been tried yet (stored in $\Theta_{\text{backtrack}}$), the node updates its children (i.e., kills the open ones, and wakes up the others) and sends them a new proposition of substitution. In this case, the node remains at the same agreement layer (Lines 6-9).

Recall that the agreement mechanism starts with the first answer of each child. However, a common agreement can appear in a subsequent step, thus the need to allow children to pursue their proof search deeper. In order to do this, the function *isConstrained* checks if the node is currently trying a substitution on its children. If so, and considering that no backtracking option is available, it means that the node has run out of potential candidates for this layer. Thus, the children are killed or awakened and the process sends them a message ($\emptyset, \Theta_{\text{forbidden}}$) to resume their proof search locally, forbidding the previously tried substitutions stored in $\Theta_{\text{forbidden}}$ (Lines 10-13).

This mechanism leads to either new substitutions or an open child, which invalidate definitively the previously chosen substitution. Consequently, if no solution can be found even without constraint on this layer, it means that the choices made at the previous layer are wrong. Thus, all the children of this layer are killed and the previous layer is woken up. An arbitrary substitution from this layer is then sent to the children, or if not, they resume their proof search without constraint (Lines 14-21).

The second main case appears when all the children have returned a substitution.

At this point, the parent undertakes the task of harmonizing them, exploring all possibilities stored in the answer structure (Line 2). If a common substitution σ is found by all the children (Line 29), it is sent to the parent (Line 30). The untried substitutions, stored in $\Theta_{\text{backtrack}}$ (Line 31), are also reported to be (possibly) tried later on this layer.

Otherwise, the agreement phase unfolds in layers: a substitution $\sigma' \in \mathit{subst}$ is picked arbitrarily (Line 34) among $\Theta_{\text{backtrack}}$ and sent to all the children (which are at that point executing *waitForParent*) to restart their proof attempts (Line 35), creating twins in turn. With the additional constraint of σ', the new proof attempts may fail, hence the necessity for backtracking among candidate substitutions $\Theta_{\text{backtrack}}$ (Line 37).

Subsequently, the new children thus produced can either readily agree, disagree, or return complementary substitutions. All the children involved in the latter are reclassified as *active* and undergo a subsequent layer of agreement, in which all the substitutions tried are consistent with the initially chosen one. On the other hand, the second case means that the applied substitution is a "bad" choice and the process backtracks, thus trying one of the other substitutions previously sent and triggering the reactivation of some previously sleeping children. This mechanism is repeated until all the children agree.

3.2.4 A Better Handling of Fairness Issues

This section applies the previous procedure to the *select pair* and *select mode* examples, showing how concurrency helps to ensure fairness. Figure 3.10 and Figure 3.12 use the same formalism to represent a proof search. It describes the parent process, in the top box, and below, the two child processes created upon application of the β-rule. Dotted lines separate successive states of a process (i.e., Procedures 3, 4 and 5 seen above), while arrows and boxes represent substitution exchanges. The number above each arrow indicates the chronology of the interactions.

Select Pair Figure 3.10 and 3.11 illustrate the interactions between processes for the *select pair* problem described in Figure 3.3b. In the former example, the problems came from b always being chosen instead of a. With the concurrent procedure, after both children have returned a substitution (1), the parent arbitrarily chooses one of them, starting with $X \mapsto b$, and sends it to the children (2). Since this substitution prevents closure in the right branch (3), the parent later backtracks, discarding the work done with the previous substitution, and sends the other substitution $X \mapsto a$ (4), allowing both children (5) and then the parent to close successfully.

Select Mode Figure 3.12 and 3.13 illustrate the interactions between processes for the *select mode* problem described in Figure 3.4b and Figure 3.4c. In the former example, the problems came from $R(X)$ never being developed, because of the closure rule being applied eagerly. With the concurrent procedure, after both children have returned a substitution (1), the parent arbitrarily chooses one of them, starting with $X \mapsto b$, and sends it to the children (2). Since this substitution prevents closure in the right branch (3), the parent later backtracks and sends the other substitution $X \mapsto a$

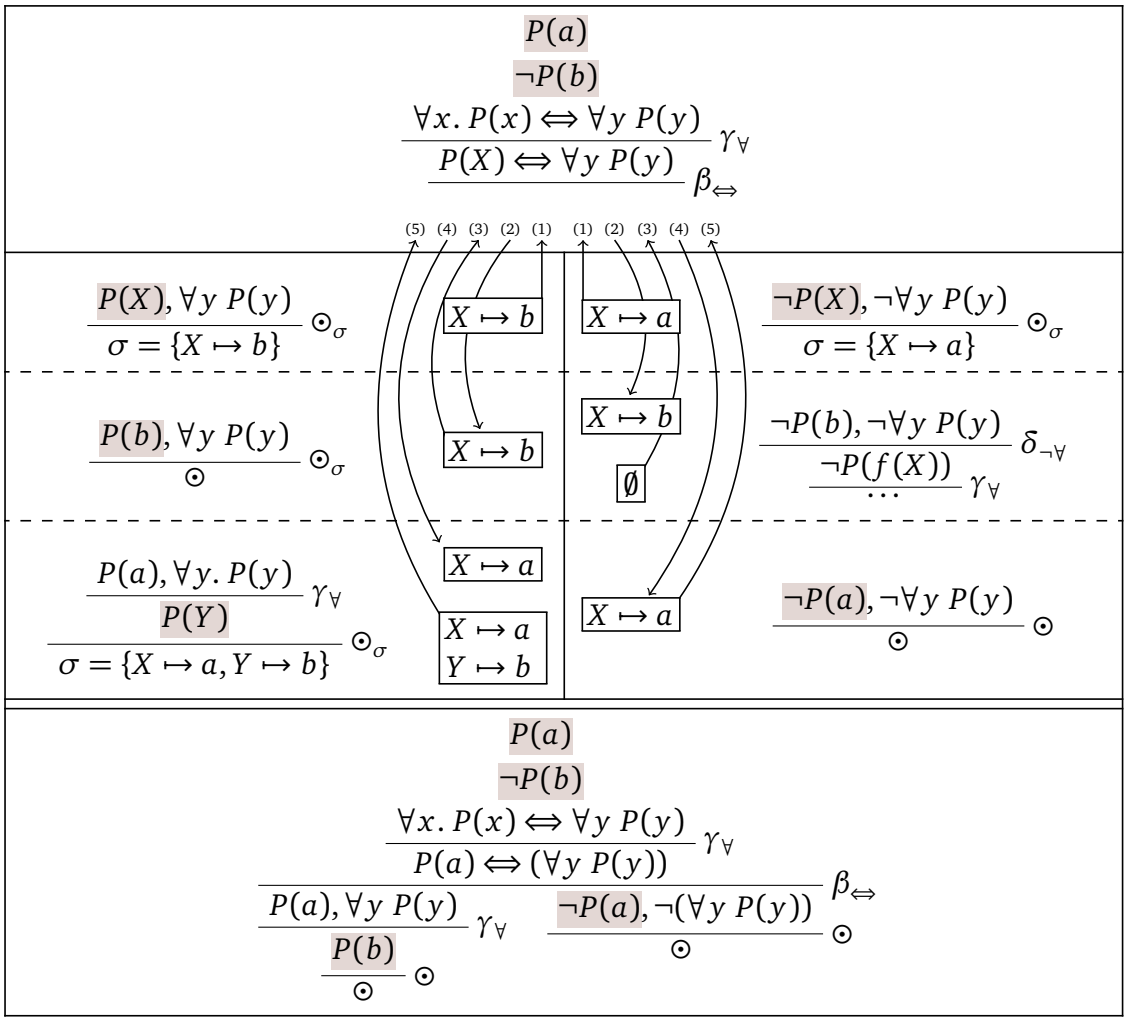


Figure 3.10: Proof search and resulting proof for $P(a), \neg P(b), \forall x.\, P(x) \Leftrightarrow \forall y\, P(y)$.

(4), which cannot close the children either (5). Ultimately, the two previously tried substitutions $X \mapsto a$ and $X \mapsto b$ are forbidden (6), and the proof search resumes with X as, again, a free variable. Thus, $R(X)$ can be developed and a contradiction can be found with $R(c)$, leading to the closing substitution $X \mapsto c$ (7).

3.3 Conclusion

We have developed a proof-search procedure for free-variable tableaux that effectively addresses many common fairness issues. The concurrent nature of this procedure enables branches to be explored in parallel, effectively solving the *select pair* problem, avoiding unfairness that can arise from sequential exploration.

In addition to the concurrent computation of branches, the procedure also overcomes the *select mode* problem. This problem typically occurs when the same substitution is repeatedly chosen and applied, leading to inefficiency and potentially incompleteness. To tackle this issue, the procedure incorporates a mechanism for for-

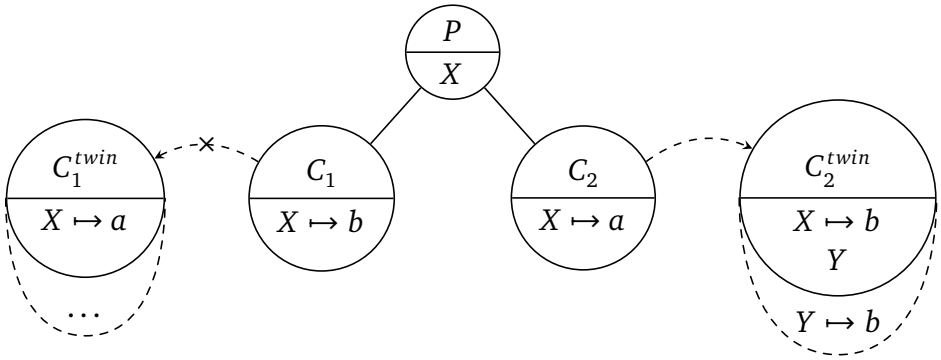


Figure 3.11: Process view of the proof search for $P(a), \neg P(b), \forall x.\, P(x) \Leftrightarrow \forall y\, P(y)$.

bidding already tested substitutions. By keeping track of substitutions and disallowing their reuse, the procedure avoids redundant exploration and improves efficiency.

The proof-search procedure and its associated techniques have been implemented in a tool called Goéland, described in Chapter 6. The details of the implementation, features, and usage of Goéland are likewise discussed throughout this thesis, providing further insights into the practical application of the developed proof-search procedure.

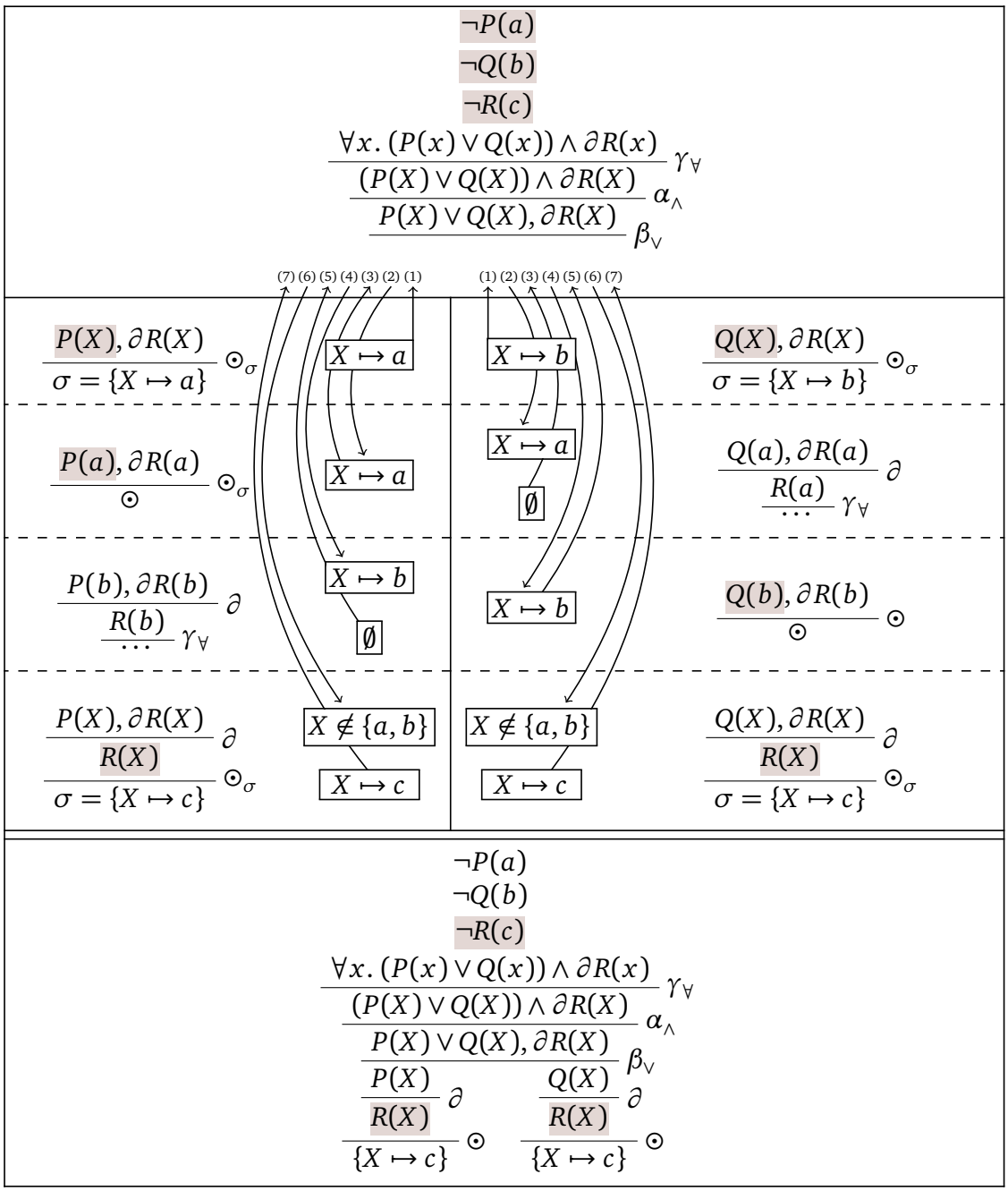


Figure 3.12: Proof search and resulting proof of $\neg P(a), \neg Q(b), \neg R(c), \forall x.\,(P(x) \vee Q(x)) \wedge \partial R(x)$.

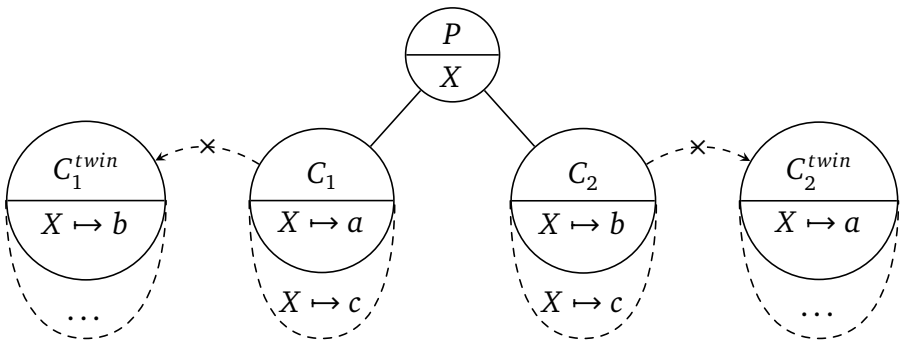


Figure 3.13: Process view of the proof search for $\neg P(a), \neg Q(b), \neg R(c), \forall x.\, (P(x) \vee Q(x)) \wedge \partial R(x)$.

Chapter 4

A Complete Proof-Search Procedure for Free-Variable Tableaux with Eager Closure

Contents

As presented in Chapter 1 and Chapter 2, the *destructive* nature of free-variable tableaux and their closure rules can lead to some fairness, and thus completeness, issues in a proof-search procedure. Nevertheless, destructive instantiation is theoretically harmless for the usual first-order tableau calculi, as it maintains proof confluency: if T can be closed, so $\sigma(T)$ can. Most textbooks on tableaux (e.g., [94, 124]) present eager closure (as defined in Chapter 1) as the standard closure rule for free-variable tableaux. Indeed, by closing a branch as soon as possible, we can hope to save resources, such as time or memory. The destructive version also avoids redoing the work, in contrast with the non-destructive version.

However, proving the completeness of a proof-search procedure with eager closure remains difficult. The main obstacle is the non-monotonicity of procedures that use backtracking. Indeed, standard completeness proofs [124] for tableau construction procedure typically proceed by considering the (infinite) proof tree that would occur if the procedure did not terminate on an unsatisfiable formula, yielding a counter-model and a refutation. For proof-search procedures with backtracking, it is not obvious how to construct this infinite derivation.

This chapter aims to give a completeness proof of the tableau-based procedure with eager closure introduced in Chapter 3. Section 4.1 presents the basic notions and context of the proof, and in particular the key notion of comparison between proof trees. Then, Section 4.2 uses this concept to prove the completeness of the

proof-search procedure introduced in Chapter 3 and implemented in the Goéland tool.

4.1 Proof Tree and Characteristics of the Proof Search

In order to understand the behavior of our method, we present the structure of a proof tree, along with an extension of the states a branch can be in, which was previously defined in Section 3.2.1. Since the tableau method allows us to reintroduce formulas, we also consider a γ-rule application limit l which binds the maximum number of terms and substitutions that can be generated with respect to l.

4.1.1 Structure of a Proof Tree and Mappings

In this chapter, we use the notion of *tableau* defined in Section 1.2, which is represented as a pair (T, σ) of a *tree* T whose nodes are decorated with sets of formulas, and a *substitution* σ over the free variables of these formulas. In a tree, we call a path from the root to a leaf a *branch*, and a path from the root to a node an *initial segment*. An initial segment S' of a tree T' is an *extension* (resp. a *strict extension*) of an initial segment S of a tree T if the formulas occurring in S are a subset (resp. a strict subset) of those occurring in S'. This is denoted $S \sqsubseteq S'$ (resp. $S \sqsubset S'$). $\sqsubseteq$ is a partial order relation, and $\sqsubset$ a strict partial order. These notions are illustrated in Figure 4.1.

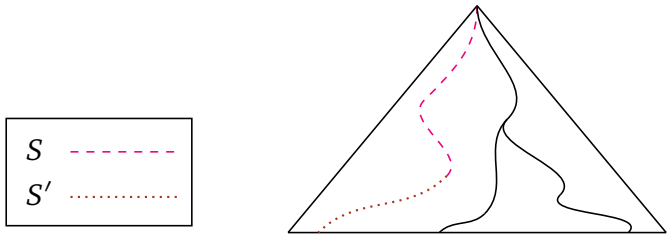


Figure 4.1: S is an initial segment, S' is a branch, and $S \sqsubseteq S'$.

We define the notion of *mapping* to identify and compare branches in two different tableaux and establish an order relation over these mappings. The idea is, starting from an open branch in a proof tree, to find the "equivalent branch" in the other proof tree, and track its evolution w.r.t. the reference branch.

Definition 4.1: Mapping (Figure 4.2)

> Let (T, σ) and (T', σ') be two tableaux derived from F. A mapping m from T' to T is a function that associates all $B' \in \text{open}_{\sigma'}(T')$ to an initial segment in T such that $m(B') \sqsubseteq B'$ (up to renaming of free variables).

Note that only open branches of T' are mapped, at least to the root. Intuitively, we want to use this notion to map a tableau generated by the procedure to a closed one with the same root formula.

We denote by $\text{img}(m)$ the multiset image of a mapping m and $\text{mult}_m(B)$ the number of occurrences of a branch B in that image. Note that only open branches of T' are mapped, in particular, if T is a closed tableau and m one of the "most extended" mapping, $\text{img}(m)$ effectively contains the remaining formulas needed to close T'.

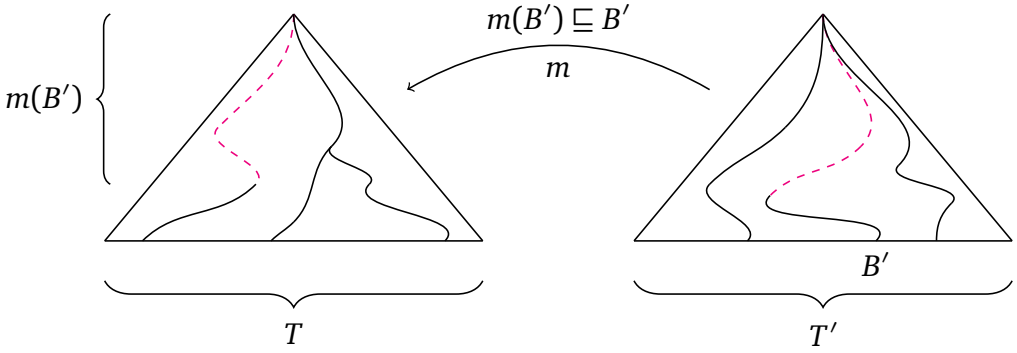


Figure 4.2: The branch B' is mapped to the initial segment $m(B')$, which means B' contains at least all the formulas of $m(B')$.

Definition 4.2: Mappings Ordering (Figure 4.3)

Let (T, σ) be a tableau. We define the order $<_{Map}$ over mappings for tableaux derived from T such that for any T' and T'' and any mapping m from T' to T and m' from T'' to T, $m' <_{Map} m$ if and only if:

- $\text{img}(m') \neq \text{img}(m)$
- for any B' such that $\text{mult}_{m'}(B') > \text{mult}_m(B')$, there exists B such that $B \sqsubset B'$ and $\text{mult}_{m'}(B) < \text{mult}_m(B)$.

$<_{Map}$ compares the images of mappings according to the multiset extension [110, 152] of the relation $\sqsubset^{-1}$ over branches. As such, it is easy to show that $<_{Map}$ is a strict partial order. Furthermore, because elements of the images are within the finite set of initial segments of T, $\sqsubset^{-1}$ is well-founded, and so is $<_{Map}$.

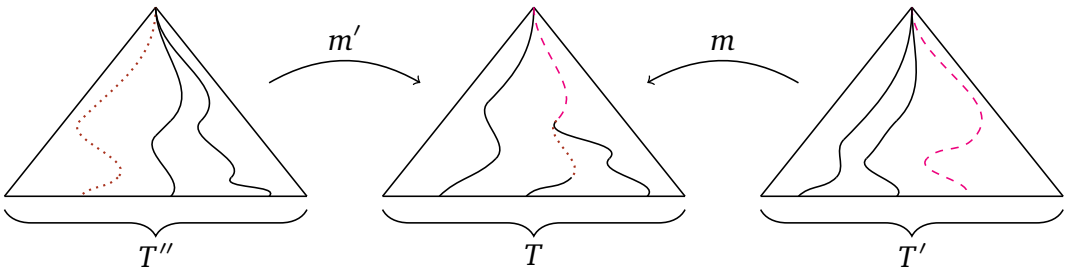


Figure 4.3: Mapping ordering between m and m' such that $m' <_{Map} m$.

Example 4.1: Mapping Ordering

Assume that branches A', B' are respective strict extensions of A, B ($A \sqsubset A', \dots$). Given mappings m with $\text{img}(m) = \{A, A, B\}$ and m' with $\text{img}(m') = \{A', A', A, B'\}$, we have $m' <_{Map} m$. Two mappings with images $\{A\}$ and $\{B\}$ are incomparable.

Definition 4.3: Mapped Tableau

Consider (T, σ) and (T', σ') two tableaux. Among the possible mappings from T to T', there is set a of minimal ones, that is to say, there exists a set of mapping M s.t. for $m \in M$, there exists no mapping m' s.t. $m' <_{Map} m$. A pair composed of the tableau (T, σ) and a mapping $m \in M$ is called a *mapped tableau*.

This definition allows us to consider only the "most extended" mapping, and to avoid dealing with irrelevant ones (for example, a mapping that maps every branch to the root).

4.1.2 γ-rule Application Limit and Higher Bound

In order to prove the termination of the procedure, we need to specify two limits related to the γ-rule application limit l. The first one is called $r_{\max}(T, l)$ and corresponds to the number of rules that can be generated from a tableau with limit l. The second one is denoted $s_{\max}(T, l)$ and expresses the number of substitutions that can be generated from a branch of a tableau with limit l. Although we can intuitively say that these numbers are bounded, we nonetheless want to define them in detail.

Lemma 4.4: Upper Bound for Substitutions per Branch (Figure 4.4)

Let $s_{\max}(T, l)$ be the number of substitutions that can be generated from a tableau T rooted in a set of formulas F with a γ-rule application limit l. A non-optimal upper bound for $s_{\max}(T, l)$ can be computed thanks to the following variables:

- t: the number of ground terms in F.
- d: the number of δ-rule in F.
- g: the number of γ-rule in F.

Let $n = (t + d + l^g)^{(l^g)}$. Then $s_{\max}(T, l)$ is bounded by $\mathcal{O}((n+1)!)$.

Proof. First of all, let us exhibit the number of values that can be assigned to a variable. Since a γ-rule can generate other γ-rules, with their own γ-rule application limit l, the number of potential free variables i l^g. Then, a free variable can be mapped to:

- a ground term;
- a Skolem term;
- another free variable.

Thus, the number of potential candidates for one free variable is bounded by $t+d+l^g$.

Consider now that a substitution contains at most the number of free variables generable. Thus, the size of the larger generable substitution is bounded by $(t+d+l^g)^{l^g}$.

Finally, since a substitution does not necessarily include all the free variables, we have to take into account the combinatorics. Thus, the number of potential generable substitutions can be upper bounded by:

$$\sum_{k=0}^{n}\binom{n}{k}=\sum_{k=0}^{n}\frac{n!}{k!(n-k)!}<\sum_{k=0}^{n}n!<(n+1)!$$

□

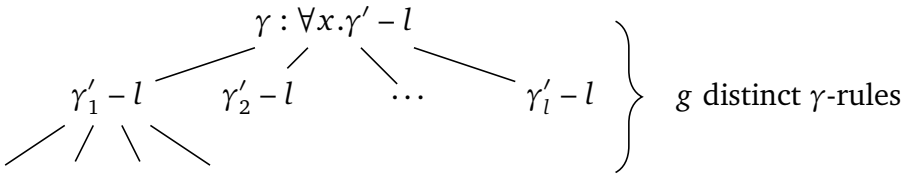


Figure 4.4: Generable free variables for one branch given a limit l.

Lemma 4.5: Upper Bound for Applicable Rules in a Tableau (Figure 4.5)

Let $r_{\max}(T,l)$ be the number of rules that can be generated from a tableau rooted in a set of formulas F with a γ-rule application limit l. A non-optimal higher bound for $r_{\max}(T,l)$ can be computed thanks to the following variables:

- c: the number of connectives in F
- q: the number of quantifiers in F

Then $r_{\max}(T,l)$ is bounded by $\mathcal{O}(2^{(l^q\times c)+1}\times s_{\max}(T,l))$.

Proof. First of all, recall that every time a β-rule is applied, the remaining formulas have to be duplicated. Thus, the worst case happens when the tableau contains only β-formulas, or at least when those formulas were computed before the others.

Moreover, one can imagine the case where all the quantifiers are universal, and generate β-formulas only. Thus, the worst case arises when applying all the γ-rules, generating l^q new formulas, with $l^q\times c$ β-connectives.

At this time, applying all the β-rules results in generating $2^{(l^q\times c)}-1$ nodes and $2^{(l^q\times c)-1}$ leaves. Then, in the worst case, each substitution can be tried at

most $s_{\max}(T,l)$ on a leaf, which is equivalent to applying $2^{(l^q \times c)-1} \times s_{\max}(T,l) < 2^{(l^q \times c)} \times s_{\max}(T,l)$ closure rules.

Finally, the number of rules applicable is bounded by:

$$\mathcal{O}(2^{(l^q \times c)+1} \times s_{\max}(T,l))$$

□

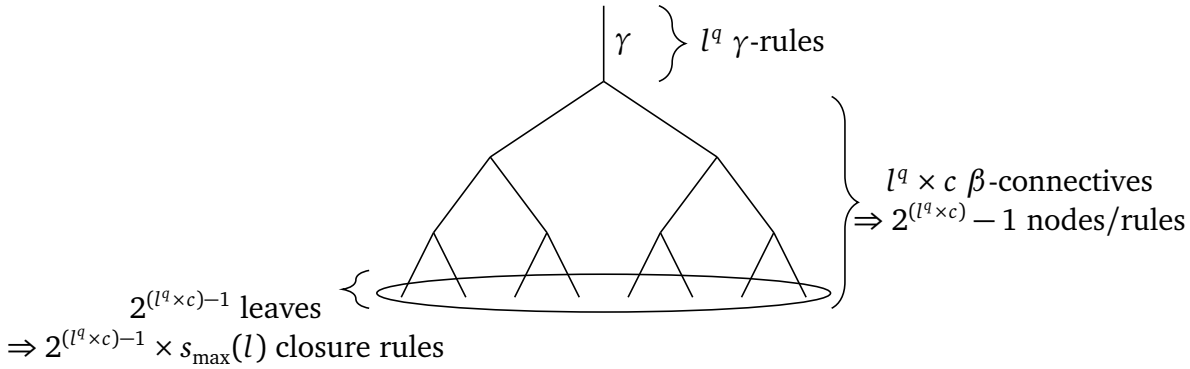


Figure 4.5: Upper bound for the number of applicable rules given a limit l.

Having established a higher bound for the rules and the substitutions, this argument can be used to show that a branch treatment is finite and to explain in detail the behavior of the branch when it reaches its limit.

4.1.3 Canonicity and k-Completeness

To prove the completeness of the procedure, we need to better characterize the branches of a tableau. The notations used here to describe a branch state correspond to those of Subsection 3.2.1. Moreover, to facilitate the comparison of two branches, we define the notion of *canonical substitution*, as well as two new branch states: *canonically instantiated* and *l-complete*.

Definition 4.6: k-Complete Branch

> For a given $k \in \mathbb{N}$, a branch B is *k-complete* if every applicable rule on formulas occurring in B has been applied, up to k application of the γ-rules.

This definition considers a branch until a certain limit of γ-rule application. It finitely approximates the more usual notion of a complete branch for free-variable tableaux, that corresponds to an ∞-complete branch formed as the directed union of k-complete branches extending one another.

Definition 4.7: Canonical Branches and Substitutions

Let (T, σ) be a closed tableau rooted in F, a tableau (T', σ') and a mapping m from T' to T. We straightforwardly extend the notion of branch mapping to variable mapping, i.e., we consider $m(X)$ to be the variable X modulo alpha-conversion in T.
A branch B' of T' is *canonically instantiated* if T' is a mapped tableau, and for any free variable X occurring both in B' and σ', $\sigma'(X)$ is unifiable with $\sigma(m(X))$. The substitution σ reduced to the free variables of B is also called *canonical*.

This concept brings together two tableaux whose substitutions do not conflict with one another.

4.2 Completeness of the Proof-Search Procedure

Our approach relies on a given arbitrary proof, which is used as a reference. We start by studying the behavior of the procedure in specific cases, before switching to the general case. In detail, we describe the implications of the l-completeness for a branch, as well as the agreement mechanism and its layers. Most proofs of completeness for tableau construction procedures rely on a notion of *fairness* (e.g., [124]). We avoid this notion, which is difficult to define for non-monotonic tableau construction procedures. A complete procedure is also necessarily fair in some sense since it must be able to retrace the steps leading to an arbitrary closed tableau.

Across this section, we fix a given reference closed tableau (T, σ) rooted in a formula F. All proof trees will be mostly implicitly rooted in F and mapped to T along Definition 4.1. Among the possible mappings from a tableau to (T, σ), we will consider the minimal ones, and thus the resulting tableau is mapped (Definition 4.3). We also denote by l the number of applications of the most reused γ-rule in T.

4.2.1 l-Completeness Behaviors

This section uses the notion of k-completeness by applying it to l. In particular, we observe that there is no branch in T which can be an extension of an l-complete branch. When applying the procedure of Chapter 3 on F, it incrementally increases its γ-rule application limit by iterative deepening and restarts the proof search, until one of the following two things happens: either it finds a proof or it reaches a limit $l' \geq l$. The former case takes a finite amount of steps and immediately solves positively the completeness question. So we focus on the latter case and assume that the procedure is applied with a limit of at least l.

Lemma 4.8: Layering of the Tableaux

> Let T' be a tableau generated by the procedure and B' a branch where some γ-formula has been reintroduced $l+1$ times. Then B' is l-complete.

Proof. By the priority order of the proof-search procedure, a γ-rule is triggered for the $(l+1)$-th time only if there is no more α-δ-β rule to apply, and there is no γ-rule left that has been introduced less than l times. □

Corollary: At most $r_{\max}(T,l)$ rules before l-completeness

> If on a branch B' of a tableau T rooted in F and generated by the procedure, $r_{\max}(T,l)$ rules have been applied, then B' is l-complete.

Proof. If some universal formula has been reintroduced $l' > l$ times, B' is l-complete by Lemma 4.8. Otherwise, by definition of $r_{\max}(T,l)$ the maximal number of rules within the limit l has been applied, therefore B' is l-complete as well. □

Definition 4.9: Fully Mapped Branches and Subtrees

> Let T' be a mapped tableau, and m be a mapping from T' to T. A branch $B' \in T'$ is said to be *fully mapped* if $m(B')$ is minimal, that is to say, $m(B')$ is a branch of T. A subtree of T' is fully mapped if each of its branches is fully mapped.

Note that branches of T go, by definition of a branch, from the root to the leaves and therefore are closed by hypothesis on T.

Lemma 4.10: l-Complete Branches are Fully Mapped

> Let T' be a mapped tableau, and B' an l-complete branch of T'. Then the mapping of B' is full.

Proof. Let m be the (minimal by Definition 4.3) mapping from T' to T. Recall that by definition of l-completeness, all the rules have been applied in B', up to l application of a γ-rule per γ-formula. Assume that the initial segment $S = m(B')$ is not a branch, and consider its extension(s) by the next rule applied, in T, to $G \in S$. Since $S \sqsubseteq B'$ and because l is the maximum number of γ-rule applications in T, this rule has also been applied to $G \in B'$. One of the children of S, say B_{ext}, is included in B' and we can extend the mapping m to $m\{B' \mapsto B_{\text{ext}}\} <_{Map} m$. This contradicts our assumption of minimality for m, therefore B' is fully mapped. □

We can remark that the substitutions applied to B' and $m(B')$ may differ, as they are not taken into account in mappings.

4.2.2 Agreement Mechanism and Completeness

This section explains the agreement mechanism described in Chapter 3, which involves twins and agreement layers and proves the completeness of the proof-search procedure for free-variable tableaux with eager closure introduced in the same chapter.

Lemma 4.11: Finite Number of Nested Agreement Layers

> There is a finite number of nested agreement layers.

Proof. Let us consider a node or a layer that has n children. To create a nested agreement layer, all the children must have answered a substitution. Then, a substitution σ is picked among the answers and sent to the nodes that have not answered σ during the previous agreement phase. This leads to the creation of twins for the nodes concerned. Since σ was returned by at least one child, the new deeper agreement layer contains at most $n-1$ children. Ultimately, for a node with n children, the agreement layer at depth k contains at most $n-k$ children and the maximal nesting depth cannot exceed n. □

Lemma 4.12: Forbidden for the Subsequent Agreement Layers

> Let us consider a layer L, and a substitution σ that has been forbidden by L. σ is forbidden for all the subsequent agreement layers, that are descendants of L, and cannot be submitted again in one of those layers.

Proof. By the procedure, when a substitution is forbidden in the children of a layer, it remains forbidden in any twin of these children. Forbidden substitutions can only be discarded in case the layer itself is open and discarded. Therefore it cannot be returned as a solution, and neither can a substitution that refines it. □

Definition 4.13: Termination for a Node

> Termination means that a node sends an answer to its parent: either the node finds an agreement between its children, or it remains open without a backtracking option, that is to say, at least one child is open without a backtracking option.

With this definition, the termination of the root node or procedure termination means tableau termination.

Lemma 4.14: Termination

Given a γ-rule application limit l, the procedure of Chapter 3 terminates.

Proof. Remember that, within a given tableau and with a limit l, there is only a finite number of available substitutions that can unify at least one branch. Moreover, there is also a finite number of rules that can be applied (see Lemma 4.4 and Lemma 4.5). In this context, the case of a leaf is immediate, since it either finds a substitution or remains open. Thus, a non-branching tree terminates. Since a leaf terminates, the non-termination may only stem from the agreement mechanism.

This termination of leaves also allows us to consider the lowest node such that its children terminate but it does not: it never finds an agreement nor runs out of potential candidates for backtracking. Let us consider the deepest agreement layer of this node that does not terminate. By hypothesis, we know that all children in the layer terminate. They all must answer some set of substitutions, because if one child remains open, the agreement layer remains open as well and, so, it terminates.

We find a contradiction by induction on the number of possible substitutions, that is to say, the number of substitutions that have not yet been forbidden by the layer. If all substitutions are forbidden, then the children can return no substitution, which is a contradiction. Otherwise, it means that at least one is available. Then, it is picked and three cases can happen:

- The children unanimously accept the substitution, this directly contradicts the non-termination hypothesis.
- The children accept the substitution, but some of them generate complementary substitutions. This creates another deeper agreement layer. This layer terminates by hypothesis. If this new layer returns a substitution, it means that the original layer terminates by returning this substitution, which is a contradiction. So this new layer has to remain open and reject the substitution.
- At least one child remains open, that is to say, the substitution is rejected.

In all cases, we see that any chosen substitution is rejected in the current layer. Since there is a finite number of possible substitutions, they are rejected in finite time and the layer has to forbid them and resume proof search (Lemma 4.12). But since at least one substitution was forbidden, we can apply the induction hypothesis for a contradiction.

Note that it makes sense to consider the deepest non-terminating agreement layer because the number of nested layers is finite (Lemma 4.11). □

Lemma 4.15: Canonical Branch Finds a Substitution

Let T' be a mapped tableau with a γ-rule application limit l and B a branch instantiated with a canonical substitution θ. If no canonical substitution is forbidden, then B sends a substitution that closes the branch and B does not forbid any canonical substitution by doing so.

Proof. By induction on the number r of rules applicable in B. The result is clear if $r = 0$: the branch is fully developed, and by Lemma 4.10 it is fully mapped. In other words, it is mapped to a leaf of T, and since no canonical substitution is forbidden, it generates, at least, the (canonical) substitution that closes the leaf of T on which B is mapped.

Otherwise, it means that all the rules have not been applied. If B is stalled or closed, it means that it has returned a substitution, so it is a successful case. If the procedure applies an expansion rule, the proof progresses w.r.t. the ordering: if the rule is α, δ, or γ, by induction hypothesis, the (unique) child of B has returned a substitution, that B can also return to its parent (in the γ case, it may prune the substitution to remove the local free variable introduced by the γ-rule).

In the last β case, B has several children $(C_i)_{1\leq i\leq n}$. The agreement mechanism steps in, so let us consider an initial layer L composed of these children. We prove, by induction on the number of children n, that L finds an agreement. We apply the original induction hypothesis on $(C_i)_{1\leq i\leq n}$. The children return the substitutions $(\sigma_k)_{1\leq k\leq m}$ (with $m \geq n$) without forbidding any canonical one. The node successively attempts each substitution σ_k on twins of $n-1$ of its children (all but the one that has proposed σ_k). Here, either:

- σ_k is canonical. By induction hypothesis, the twins will generate substitutions without forbidding a canonical substitution, and a new layer will be created, with $n-1$ (or fewer) children. By the latest induction hypothesis, they find and return an agreement, without forbidding any canonical substitution, and so does L and B.

- σ_k is non-canonical but the twins find an agreement: this is a success case, too.

- Otherwise, no σ_k is canonical and none fits all the children. A set of forbidden substitutions is sent to $(C_i)_{1\leq i\leq n}$, prohibiting the non-canonical $(\sigma_k)_{1\leq k\leq m}$ and prompting the children to generate new substitutions. These nodes still comply with the original induction hypothesis so they do generate new candidates. This happens at most $s_{\max}(T, l)$ times before we reach one of the previous cases.

□

Theorem 4.16: The Procedure of Chapter 3 is Complete

> Let us suppose that there exist a tableau (T, σ) with a limit l which is a proof for a formula F. Then, by applying the procedure with a limit l, a proof for F is generated.

Proof. The root node is a canonical branch. By Lemma 4.15, the procedure finds a proof for F. □

4.3 Conclusion

We have proven the completeness of the non-monotonic tableau construction procedure introduced in Chapter 3. This method works with a given proof, used as a reference, and relies on the notion of *mapping*, which allows a proof-tree comparison. To apply our results to a procedure that relies on iterative deepening, we have also enhanced the description of the branch states with respect to a certain limit.

In detail, the notion of mapping establishes a correspondence between branches of two proof trees, allowing us to analyze and compare their structures. By examining the mappings at each step of the procedure, we can track the evolution and fluctuation of the proof search towards the desired proof.

In order to give a bound to the proof-search procedure, we have specified the notion of *l-completeness*, which permits us to regard a tree as completely developed within a specific threshold. We have also examined the specificity of the proof-search procedure, i.e., its agreement mechanism, to finally prove its completeness.

The concepts and conditions used in this proof fit well with those of backtrack and eager closure, which are standard strategies for tableau-based theorem provers. Thus, based on those specifications, we plan to develop a general method to prove the completeness of proof-search procedures for free-variable tableaux with eager closure.

Chapter 5
Handling Theories in Tableau-Based Automated Reasoning Methods

Contents

In recent years, reasoning efficiently with theories has been of growing interest in the community of automated theorem proving. This surge can be attributed to the increasing number of real-world problems involving heavy theories, such as the BWare set theory [105], or to those translated into firs-oder logic from other languages, such as the proof obligations of an interactive theorem prover exported with a tool, for instance, Sledgehammer [51]. This trend has forced theorem provers to adapt themselves to these new challenges, developing new strategies to address them.

Whereas there exist efficient methods to deal with a specific domain, there is no universal approach to handle them all, and the addition of theory axioms to problem hypotheses is rarely usable in practice. Indeed, some theories can contain thousands of axioms, and the resulting increase of the size of the problem does not allow efficient reasoning, in addition to not being able to identify relevant axioms. Conversely, leveraging the domain-specific knowledge to develop efficient reasoning techniques gives better results and is currently the standard way to deal with a specific domain, especially in tableaux, for example, with arithmetic [211]. This involves the interaction between a general-purpose *foreground reasoner* and a specialized *background reasoner* designed for dealing with problems related to a particular theory. This collaboration occurs during key moments of proof search, guiding the process in the correct direction.

Among these theories, equality stands out due to its prevalence and its expressiveness. Yet, handling equality reasoning in tableaux is notably challenging, leading

to a wide range of studies and techniques [28, 29, 128] aimed toward this goal: handling equality in a tableau-based proof-search procedure. While some satisfactory solutions have been found [103], the pursuit of more efficient equality handling techniques continues [15].

Considering that each theory may require distinct treatment, one could envision having a dedicated background reasoner for each theory, akin to the approach taken with equality reasoning. This strategy is undoubtedly efficient but very time-consuming and not very adaptable to new theories. In contrast, the concept of an extensible background reasoner capable of handling a multitude of theories has gained traction. This approach, known as *Deduction Modulo Theory* [115], extends predicate calculus, enabling the rewriting of terms and propositions. It proves suitable for proof search in axiomatic theories, converting axioms into rewrite rules. By automating this transformation and meticulously managing interactions with the proof search process, an efficient generic background reasoner can be established, resulting in a polyvalent automated theorem prover.

The goal of this chapter is twofold: introduce state-of-the-art techniques to handle theories in free-variable tableaux and discuss their implementation into an automated concurrent tableau-based theorem prover. It starts by examining the theory of equality in Section 5.1, and progresses toward a a complete equality reasoning procedure. Following this, a universal approach for handling any axiomatized theory, the deduction modulo theory, is described in Section 5.2. This method outlines how the set of axioms representing a theory can be managed without requiring additional resources. Since these methods have already been studied in the literature, our contribution to this field is composed by their implementation in the context of a concurrent tableau proof search, highlighting its pivotal interactions with the main procedure.

5.1 Equality Reasoning

One of the main goals of automated deduction is to efficiently handle first-order logic with equality. The incorporation of proficient equality reasoning into tableau and sequent calculi poses a long-standing challenge, resulting in a plethora of theoretically captivating but surprisingly few practically gratifying solutions. Given its potency, expressive nature, and ubiquity in real-world problems, the inclusion of equality reasoning is essential for any robust reasoning method.

In light of its distinctive characteristics, several techniques have been designed to directly address equality within the framework of tableau-based calculi. Noteworthy examples include the equality elimination method [100], the introduction of new tableau rules into the calculus [135], and the application of transformations from first-order logic with equality to the corresponding form without equality in the input formulas [12, 68].

However, since equality remains a theory, conventional approaches designed for managing theories can also be harnessed to integrate equality reasoning as a background reasoner. Consequently, most strategies for handling equality can be viewed as specific instances of general methods for theory reasoning within semantic

tableaux. Among these approaches, a family of methods employing incomplete unification procedures in such a way that an overall complete first-order calculus is obtained has been developed: the rigid E-unification.

This section focuses on presenting how rigid E-unification, as defined in [129], can be used to efficiently handle equality in free-variable tableaux. It begins by establishing the foundational principles of equality reasoning, subsequently illustrating the process of extracting a problem from a tableau as well as an approach for its resolution. Finally, it discusses its implementation and its interaction with a concurrent proof-search procedure.

5.1.1 Equality Reasoning in Tableau-Based Systems

To formally initiate the definition of equality, it becomes necessary to distinguish between syntactic meta-level equality and semantic equality, which pertains to the theory of equality addressed here. In other words, when considering two terms t and t', $t = t$ holds true as they are syntactically identical terms, and $t = t'$ do not. Conversely, $t \approx t'$ implies that t and t' are semantically equivalent and can be interchanged while maintaining the semantic integrity of the formula. Within this section, the latter notion will be referred to as *equality* (and its negation as *inequality*).

The $\mathcal{E}$ Theory Intuitively, the concept of equality implies that two distinct terms, even syntactically different, can be exchanged without altering the truth value of the formula. Through these symbols, the theory of equality can be defined by its fundamental axioms:

Definition 5.1: Equality Theory

The Equality theory, also denoted $\mathcal{E}$, is composed of the following axioms:

(1) (reflexivity) $x \approx x$

(2) (symmetry) $x \approx y \Rightarrow y \approx x$

(3) (transitivity) $x \approx y \wedge y \approx z \Rightarrow x \approx z$

(4) (monotonicity for function symbols) for all function symbols $f \in \mathcal{S}_F^n$:
$x_1 \approx y_1 \wedge \ldots \wedge x_n \approx y_n \Rightarrow f(x_1, \ldots, x_n) \approx f(y_1, \ldots, y_n)$

(5) (monotonicity for predicate symbols) for all predicate symbols $P \in \mathcal{S}_P^n$:
$x_1 \approx y_1 \wedge \ldots \wedge x_n \approx y_n \Rightarrow P(x_1, \ldots, x_n) \approx P(y_1, \ldots, y_n)$

A Tableau Proof Search Including Equality Axioms Figure 5.1 presents a first-order problem with equality axioms in a tableau proof-search procedure. Similar to usual tableaux, all the branches need to be closed in order to obtain a proof, possibly with the help of a global unifier. In this specific example, the tableau splits into two

branches, B_1 and B_2. The substitution $\{X \mapsto a\}$ immediately solves B_2, thanks to $\neg a \approx a$. A solution is also found for B_1 with the application of equality reasoning on $P(g(g(a)), b)$ and $\neg P(a, c)$, thanks to the equalities $b \approx c$, $f(X) \approx g(X)$ and $g(f(Y)) \approx Y$. In details, $b \approx c$ allows the respective second elements of the predicates to match, whereas $g(g(a))$ is first transformed into $g(f(a))$ due to $f(X) \approx g(X)$ and then into a thanks to $g(f(Y)) \approx Y$, leading to $P(a, c)$ and thus to the contradiction with $\neg P(a, c)$. Since the closure on B_1 does not require any substitution, $\{X \mapsto a\}$ is a global closure for the tableau.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x.\, g(x) \approx f(x) \vee \neg(x \approx a)}{\forall y.\, g(f(y)) \approx y}}{b \approx c}}{P(g(g(a)), b)}}{\neg P(a, c)}}{g(f(Y)) \approx Y}\,\gamma_\forall}{(g(X) \approx f(X)) \vee \neg(X \approx a)}\,\gamma_\forall}{\begin{array}{cc} g(X) \approx f(X) & \neg(X \approx a) \\ B_1 & B_2 \end{array}}\,\beta_\vee$$

Figure 5.1: Equality problem.

Equality Reasoning Challenges While these types of problems may appear manageable to human intuition, the computational aspect is considerably more intricate due to the difficulty machines face in deducing the "correct" equality to apply. The task of selecting the appropriate equality replacement at the outset of the proof search can result in unfavorable choices, leading to a lot of useless proof-search steps before potentially reevaluating the decision. Additionally, the straightforward approach of exhaustively attempting all equality replacements without restriction can trigger an unending sequence of term generation, rendering it unfeasible in practice.

Maintaining the coherence of the tableau presents yet another substantial challenge, akin to the complexities of managing substitutions and their interactions. Specifically, when an equation such as $X \approx a$ is part of the tableau and X is affected by a substitution, it becomes imperative to ensure that such alterations do not adversely impact other branches or equations involving the variable X.

Reasoning Method to Handle Equality in Tableaux There are two primary techniques for handling equality in semantic tableaux: *partial equality reasoning*, the more straightforward method consisting of adding new rules to the tableau calculus, and *total equality reasoning*, which hinges on E-Unification, a decision procedure that determines branch closure without additional expansion rules.

Efforts to transform more sophisticated and efficient methods, such as completion-based approaches, into simple tableau expansion rules face difficulties. The shared problem among these partial reasoning methods, founded on extra tableau expansion rules, is that equalities can be applied without any restriction. The symmetry of these rules leads to extensive search space, making it challenging to solve even relatively straightforward problems in a reasonable amount of time [32].

In contrast, the utilization of *total equality reasoning*, which avoids the addition of equality expansion rules, transforms the task of finding a closing substitution in a tableau branch into solving an E-unification problem. This approach permits the use of various algorithms for E-unification problem-solving. Additionally, [33] demonstrated that E-unification-based methods significantly outperform those based on additional rules.

The main idea behind this concept is to define an E-unification problem comprising a pair of complementary literals and an equality set E, which is then solved by unifying the two literals using the available equalities in the branch. Multiple algorithms facilitate the matching process, contributing to the efficiency of this approach compared to the partial one. Moreover, different forms of E-unification exist based on the type of tableau (grounded, free variables, etc.). For free variables tableaux, the corresponding problem is known as the *rigid E-unification* problem.

The importance of rigid E-unification for automated theorem proving was initially outlined in [130]. Later, [129] refined the concept of [43] to suit equational reasoning in tableaux. A complete procedure was then developed in [103], which uses E-unification to address equality reasoning in tableau-based systems.

5.1.2 Extraction of a Rigid E-Unification Problem

This kind of E-unification is called *rigid* due to its distinctive feature: contrary to classical E-unification, it only allows a variable to be assigned to a single term [129]. Then, an E-unification problem can be defined as follows:

Definition 5.2: Rigid E-Unification Problem

A rigid E-unification problem

$$\langle E, s, t \rangle$$

consists of a finite set E of equalities of the form $(l \approx r)$ and two terms s and t such that $r, l, s, t \in \mathscr{T}$.

Intuitively, this problem represents the intent to establish equality between s and t, leveraging the equalities present within E. A *solution* to a rigid E-unification problem is a substitution containing the required instantiations of free variables, which have been essential in demonstrating the equivalence of the two provided terms.

Example 5.1: Rigid E-Unification Problem

Let $\langle \{g(Y) \approx Y, g(Z) \approx f(Z)\}, g(g(a)), f(X)\rangle$ be an E-unification problem. The goal is to match $g(g(a))$ and $f(X)$. To do so, $g(g(a))$ can be reduced to $g(a)$ with the use of the first equality, w.r.t. the substitution $Y \mapsto a$. Then, $g(a)$ can be transformed into $f(a)$ following the second equality, and improving the current substitution by $\{Z \mapsto a\}$. Finally, $f(X)$ and $f(a)$ are unifiable, which leads to the final substitution $\{X \mapsto a, Y \mapsto a, Z \mapsto a\}$. Other combinations of rules also allow us to solve this problem.

An E-unification problem is the basic element for rigid E-unification. In detail, for each par of complementary literal $P(t_0, \ldots, t_n)$ and $\neg P(t'_1, \ldots, t'_n)$, we extract n sub E-unification problems in order to match the i^{th} terms together. An inequality $t \neq t'$ also leads to an E-unification problem, in which t and t' have to be matched.

Extraction of Rigid E-Unification Problems Recall the example of Figure 5.1. There are two branches, B_1 and B_2, each of them with their own equality set. In this example, the notation $E(B_i)$ represents the set of equalities of the branch i and $P(B_i)$ the set of set of rigid E-unification problems corresponding to each pair of complementary literals or inequalities.

Example 5.2: Simultaneous Rigid E-Unification Problem Corresponding to the Proof Tree in Figure 5.1

The sets of simultaneous rigid E-unification problem of the figure Figure 5.1 are the following:

- B_1 :
  - $E(B_1)$:
    - $b \approx c$
    - $g(f(Y)) \approx Y$
    - $g(X) \approx f(X)$
  - $P(B_1)$:
    - $\{\langle E(B_1), g(g(a)), a\rangle, \langle E(B_1), b, c\rangle\}$
- B_2 :
  - $E(B_2)$:
    - $b \approx c$
    - $g(f(Y)) \approx Y$
  - $P(B_2)$:
    - $\{\langle E(B_2), g(g(a)), a\rangle, \langle E(B_2), b, c\rangle\}$
    - $\{\langle E(B_2), X, a\rangle\}$

Within this instance, B_1 is associated with a single equality problem while B_2 has two of them. Moreover, B_2 comprises two distinct closure types: a literal and its opposite and the negation of an equality. The overarching objective is to identify a substitution that both B_1 and B_2 can concur upon, ensuring the closure of the entire proof tree. Once the problem is established, attention can then be directed towards exploring different strategies for effectively resolving it.

5.1.3 Handling Problems with Equality in a Tableau-Based Proof-Search Procedure

Resolution of a Rigid E-Unification Problem In [103], a method called rigid basic superposition was introduced for computing a finite (albeit incomplete) set of solutions for rigid E-unification problems. This approach adapts basic superposition (in the formulation presented in [187]) to rigid variables. Despite being incomplete, these solutions are sufficient to handle equality within rigid variable calculi. This method is improved by a set of constraints as well as a term ordering relation and relies on two rules (left and right), which influence either the equalities or the two terms to match.

Together with these rules, complete calculus for first-order logic with equality has been constructed: the $\mathcal{BSE}$ calculus. The rules are integrated into the closure rule of free-variable tableaux. This extension works as follows: each solution computed by rigid basic superposition for one of the unification problems in $P(B)$ can be used to close the branch B.

In detail, recall that for two complementary literals, a set of E-unification problems is extracted, each having the same set of equalities. For a single rigid E-unification problem, multiple solutions can arise. Thus, for each identified term match, i.e., a substitution σ such that $\sigma(s)$ and $\sigma(t)$, σ has to be compatible with the solution returned by the others E-unification problem stemming from the same predicate. To manage this, a mechanism similar to the one employed for the children management in Chapter 3: each problem searches for a solution, and then an agreement phase happens. If one problem is unable to be solved, the set of problem admit no solutions and the closure mechanism start on the next pair of predicates. Another layer of parallelization could also have occurred inside of the resolution of a problem itself: in the basic rigid superposition rules, at each step, multiple right or left rules can be applied. We prioritize the right ones [125], which are the ones leading to the closure case, as illustrated in Figure 5.2. After this selection, only one type of rule remains, and one could imagine launching them all in parallel. This obviously leads to some redundant states, which could be pruned. However, this mechanism was not implemented, since the generated goroutines would have downgraded too much the proof search in terms of performance.

To control the rule application's expansion, since one pair of complementary literals is enough to close a branch, the reasoning stops once at least one set has a solution. The state of the equality reasoning is saved, to be possibly re-opened in a subsequent step.

Integration into the Proof-Search Procedure We now study the interactions between equality reasoning and the proof-search procedure. Two main strategies

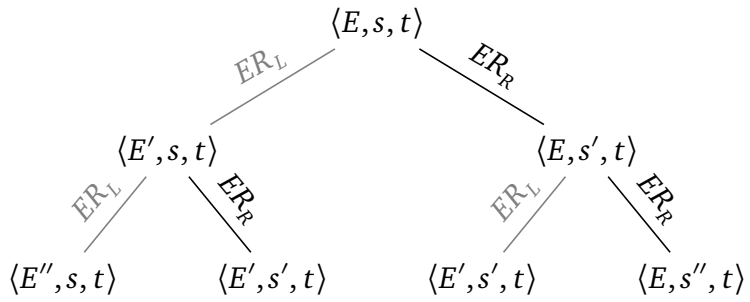


Figure 5.2: Equality rule application with optimizations.

are usually employed: solving all the rigid E-unification problems of all the branches at once, or closing branches sequentially, the first substitution found to close a branch B_i being applied to the whole tableau. If, at a later point, the closure of branch $B_j (j > i)$ becomes impossible, a backtracking process is triggered to compute supplementary closing substitutions for B_i. This approach yields a calculus that is both correct and complete, provided that the search for additional branch-closing substitutions is bounded.

The first case is referred to as *simultaneous* rigid E-unification problem. This problem takes a set of equalities and terms and attempts to find a global unifier, which results in a closure for the whole tree.

Definition 5.3: Simultaneous Rigid E-Unification Problem

A finite set

$$\{\langle E_1, s_1, t_1\rangle, \ldots, \langle E_n, s_n, t_n\rangle\}$$

of rigid E-unification problems is called a simultaneous rigid E-unification problem. A substitution σ is a solution to the simultaneous problem iff it is a solution to every component $\langle E_i, s_i, t_i\rangle (1 \leq i \leq n)$.

In simpler terms, every set of simultaneous rigid E-unification problems representing a branch must converge towards a common substitution. However, within each individual branch, the need for only one set to achieve closure prevails. This combinatorics is depicted in Figure 5.3.

Conversely, a more straightforward implementation relies on closing one branch after another, sending the closing substitution to the sibling branches, just as usual substitution management. However, when a substitution is found for a branch, it may not necessarily fit with the other branches. In such cases, revisiting the previous branch becomes necessary to explore potential new solutions. Hence the need for backtrack points in the procedure, allowing the possibility to restart all the E-unification problems that have returned a solution for a branch and reconsider a choice. By memorizing work already done, the procedure can easily resume the proof search without redoing work. However, if the proof search resumes with a substitution, the equality reasoning has to restart too, because the previous equality reasoning step may not hold anymore with this additional constraint.

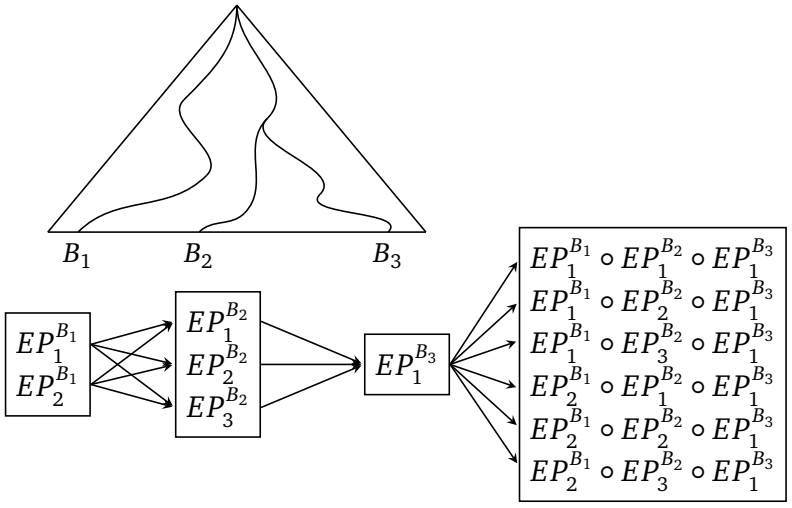


Figure 5.3: Equality reasoning combinatorics for simultaneous equality problems.

Another point intricately linked to proof search pertains to the timing of applying equality reasoning. Applying it at every step is possible, but clearly inefficient. An alternative approach involves waiting until a branch is fully developed before attempting it, or at least, waiting for a set of branches to find at least one contradiction. This solution is used for the simultaneous case. At the end of the day, a balanced approach for the per-branch-closure case could involve applying equality reasoning solely when a predicate is generated, akin to the style of closure rules. This integration would augment the closure mechanism, resulting in a sound and complete calculus for first-order logic with equality [31, 32]. In the end, equality reasoning is avoided in cases where it is irrelevant, such as when the new predicate is not part of any equalities.

Moreover, unlike other predicates, $\approx$ receives direct interpretation during the proof-search process: for any term t, $\neg(t \approx t)$ is assigned to $\bot$. This aspect is incorporated as a closure rule within the proof search.

Handling all (or multiple) branches of a tableau in parallel proves to be more efficient [32]. This enables simultaneous utilization of the information contained within the branches, obviating the need for backtracking and allowing for a constrained search space. For instance, it's often feasible to identify branches wherein only one closing substitution exists. These substitutions can be promptly applied even before other branches are closed. Although the per-branch method was originally unable to combine information coming from different branches, the use of parallelization in the proof search allows it to gain a communication level closer to the simultaneous case, and thus enhance performance.

5.2 Deduction Modulo Theory

The goal of this section is to describe a general-purpose system designed to generically handle theories and its interactions with the proof-search procedure presented in Chapter 3: the deduction modulo theory (DMT). The main idea behind this system is to transform axioms into computational rules, resulting in a congruence calculus

and a reduction in the proof-search space. The first ideas of this system appear as early as 1965 [202], with the idea of replacing axiom by deduction rules, and 1972 [199], where the author replaces standard unification with unification modulo associativity in a resolution-based system.

The issue raised is simple: an automated theorem prover might apply trivial lemmas such as the associativity of addition a great number of times, thus leading to an inefficient proof search. As heuristics and *ad-hoc* solutions do not lead to efficient results, he found that orienting the terms in the right way gives a confluent rewrite system. Consequently, the associativity axiom could be replaced by a computational rule that keeps the completeness of the original system.

Over time, this technique gained widespread recognition and started encompassing a broader range of theories. Initially, these extensions were focused on other theories such as simple type theory [114], arithmetic [117], and Zermelo's set theory [116]. Recently, its application field was extended to systematically transforming the axioms of any theory into rewrite rules as a potential optimization for automated proof-search procedures [104].

The results yielded by this approach on Zenon [65] give birth to the provers Super Zenon [153], Zenon Modulo [104], and inspired iProver Modulo [72]. This section first focuses on defining deduction modulo theory and the automatic translation of an axiom into a rewrite rule. It then delves into useful variants of deduction modulo theory in a tableau proof-search procedure and finishes by highlighting potential pitfalls in managing interactions between the procedure and the rewrite system.

5.2.1 Motivation, Definition and Rewriting

Section 5.1 has highlighted an instance of a specific background reasoner dedicated to equality reasoning. While deduction modulo theory could also handle equality in a certain way, the present version excludes the equality predicate from the reasoning and rather focuses on offering computation over propositions. Indeed, interactions with the equality reasoner presented in the previous section are not trivial and could lead to conflicts, as both mechanisms would like to work on the same set of terms. Further investigation needs to be performed before combining the two tools.

The advantages of deduction modulo theory over the native proof-search procedure can be seen fairly easily when considering the set theory's theorem $\forall a.\, a \subseteq a$ under the following axiom:

$$\forall a, b.\, a \subseteq b \iff \forall x.\, x \in a \Rightarrow x \in b$$

The tableau proof of this problem is pretty standard and developed in Figure 5.4a. As can be seen, this proof needs to explore two branches to manage the axiom corresponding to the inclusion. This case is pretty common in axiomatic theories, where some of the axioms of a theory T define predicates introduced for reasoning in T alongside their semantics. Thus, since they are equivalent, two branches need to be managed: the predicate and its definition.

This kind of axiom presents a great bottleneck for a procedure based on the tableau method, as the branching induced by disjunctions leads to an explosion of the

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall a, b.\, a \subseteq b \Leftrightarrow \forall x.\, x \in a \Rightarrow x \in b, \neg(\forall a.\, a \subseteq a)}{\neg(c \subseteq c)}\,\delta_{\neg\forall}}{c \subseteq c \Leftrightarrow \forall x.\, x \in c \Rightarrow x \in c}\,\gamma_{\forall},\gamma_{\forall}}{c \subseteq c, \forall x.\, x \in c \Rightarrow x \in c}\,\beta_{\Leftrightarrow}}{\odot}\,\odot \qquad \dfrac{\dfrac{\dfrac{\neg(c \subseteq c), \neg(\forall x.\, x \in c \Rightarrow x \in c)}{\neg(c' \in c \Rightarrow c' \in c)}\,\delta_{\neg\forall}}{c' \in c, \neg(c' \in c)}\,\alpha_{\neg\Rightarrow}}{\odot}\,\odot}$$

(a) Tableau proof that $a \subseteq a$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg(\forall a.\, a \subseteq a)}{\neg(c \subseteq c)}\,\delta_{\neg\forall}}{\neg(\forall x.\, x \in c \Rightarrow x \in c)}\; c \subseteq c \longrightarrow \forall x.\, x \in c \Rightarrow x \in c\,\delta_{\neg\forall}}{\neg(c' \in c \Rightarrow c' \in c)}\,\alpha_{\neg\Rightarrow}}{c' \in c, \neg(c' \in c)}\,\odot}{\odot}\,\odot$$

(b) Tableau proof with deduction modulo theory that $a \subseteq a$.

Figure 5.4: Comparison between a standard tableau proof and a proof that use a rewrite system thanks to deduction modulo theory.

proof-search space, necessitating the reconciliation of a substantial number of leaves. Deduction modulo theory excels in its ability to convert these branching-inducing axioms into computational rules, thereby reducing the size of the proof-search space. For example, let us take the same theorem to prove as before, i.e., $\forall a.\, a \subseteq a$, but now the set theory's axiom describing the inclusion is picked out of the proof search and transformed into the following rewrite rule:

$$A \subseteq B \longrightarrow \forall x.\, x \in A \Rightarrow x \in B$$

where A and B are free variables that await an instantiation through unification. It leads to the proof of Figure 5.4b, in which the tableau is easily closed in a few steps, without branching. With this intuition established, a formal definition of a *rewrite rule* and a *rewrite system* can be given.

Definition 5.4: Rewrite Rule

A *rewrite rule* is a pair of formulas F, G such that $\mathrm{FV}(G) \subseteq \mathrm{FV}(F)$. It is denoted $F \longrightarrow G$.

In the state of the art, a rewrite rule can ether be applied on terms or propositions. In this thesis, as only the propositional case is mentioned, we use the denomination *rewrite rule* to talk about *propositional rewrite rule*.

Definition 5.5: Rewrite System

A *rewrite system* $\mathcal{R}$ is a set of *rewrite rules*. A rewrite rule $F \longrightarrow G$ is part of a rewrite system $\mathcal{R}$ if $(F, G) \in \mathcal{R}$ and is denoted $F \longrightarrow_{\mathcal{R}} G$.

Definition 5.6: Rewriting Relation

Let $\mathcal{R}$ be a rewrite system and A, B be formulas. A rewrites to B, denoted $A \longrightarrow_{\mathcal{R}} B$ if and only if:

- either there exist a rewrite rule $F \longrightarrow_{\mathcal{R}} G$ and there exist a substitution σ such that $A = \sigma(F)$ and $B = \sigma(G)$;
- or A and B are compound formulas with the same main connective or quantifier, and exactly one pair of the corresponding subformulas rewrites one to another, the rest being identical.

For any rewrite system $\mathcal{R}$, we will denote $\longrightarrow^*_{\mathcal{R}}$ to be the reflexive and transitive closure of $\longrightarrow_{\mathcal{R}}$.

Definition 5.7: Confluence, Termination

A rewrite system $\mathcal{R}$ is said *confluent* if for a given formula F, for all formulas F_1, F_2 such that $F \longrightarrow^*_{\mathcal{R}} F_1$ and $F \longrightarrow^*_{\mathcal{R}} F_2$ there exists F' such that $F_1 \longrightarrow^*_{\mathcal{R}} F'$ and $F_2 \longrightarrow^*_{\mathcal{R}} F'$. Furthermore, $\mathcal{R}$ is said *terminating* if there does not exist an infinite chain of derivations.

It is important to note that $\longrightarrow_{\mathcal{R}}$ does not, in general, preserve the proof confluency and thus may lead to an incomplete procedure if not properly handled. Moreover, as deduction modulo theory can lead to the loss of cut-free completeness [112], we need to pay specific attention to the design of a computational system.

A question to be asked is whether to develop a manual rewrite system or to use a heuristic to automatically transform axioms into rewrite rules. The former has been investigated in [78] and has been found to be greatly efficient in a given theory (the set theory of the B method in particular). Moreover, it also allows us to study in detail the rewrite system, for instance, its confluency, its termination, or its cut-free completeness. However, this approach lacks the generality expected from deduction modulo theory in an automated prover, as even if a great effort was spent on writing a file containing the rewrite rules of numerous theories, there would surely come a time when it would be rendered useless.

The choice for a general-purpose proof-search procedure has thus been to use a heuristic to translate axioms into rewrite rules, effectively removing the translated axioms from the original problem to prove. This approach has been thoroughly investigated in [73], which outlines the main properties needed by a heuristic to

yield rewrite systems that are not naively non-terminating. The term "naively non-terminating" is used because achieving a (theoretically) terminating rewrite system without causing significant performance degradation is challenging.

For example, we should refrain from transforming an axiom expressing the commutativity of an operation, as it inevitably leads to an endless sequence of rewrite steps once triggered. Another easily avoidable pitfall when transforming axioms into rewrite rules is to prevent the rewriting of a proposition P into another proposition A if there exists a subformula of A unifiable with P. However, these solutions solely solve special cases of a more general and pathological problem, and thus do not ensure a theoretically terminating automatically-generated rewrite system. As a matter of fact, let us take two axioms of a naive set theory.

$$\forall a, b.\, a \subseteq b \Leftrightarrow a = b \vee a \subsetneq b$$
$$\forall a, b.\, a = b \Leftrightarrow a \subseteq b \wedge b \subseteq a$$

As can be seen, the two cases observed in the previous paragraph do not happen in these axioms. Thus, they can "safely" be transformed into the following rewrite system $\mathcal{R}$.

$$A \subseteq B \longrightarrow_{\mathcal{R}} A = B \vee A \subsetneq B$$
$$A = B \longrightarrow_{\mathcal{R}} A \subseteq B \wedge B \subseteq A$$

where A and B are, as explained before, free variables awaiting an instantiation by unifying these rewrite rules with formulas derived in the proof search. $\mathcal{R}$ is non-terminating, as we can exhibit the following infinite rewriting chain:

$$A \subseteq B \longrightarrow_{\mathcal{R}} A = B \vee A \subsetneq B \longrightarrow_{\mathcal{R}} (A \subseteq B \wedge B \subseteq A) \vee A \subsetneq B \longrightarrow_{\mathcal{R}} \cdots$$

In this particular case, it is still easily checkable, even though more efforts need to be made to detect the infinite potential chain. Nevertheless, in practical terms, it is inefficient to verify the termination of such systems. Instead, it is more effective to ensure termination by carefully rewriting during the proof-search procedure.

In state-of-the-art heuristics of deduction modulo theory in tableau-based theorem provers, two types of axioms are usually chosen to be translated in rewrite rules:

$$\forall \vec{x}.\, P$$
$$\forall \vec{x}.\, P \Leftrightarrow A$$

where P is an atomic proposition and A any non-literal. Furthermore, recall that by definition, the last two types of axiom can only be translated into rewrite rules if $\mathrm{FV}(P) \subseteq \mathrm{FV}(A) \subseteq \vec{x}$. The following rewrite system is thus derived:

$$\begin{array}{rcl} \forall \vec{x}.\, P & \text{becomes} & P \longrightarrow \top \\ \forall \vec{x}.\, P \Leftrightarrow A & \text{becomes} & P \longrightarrow A \end{array}$$

To manage the integration of rewrite rules into the proof-search procedure, the portion of the substitution σ of Definition 5.6 that involves free variables belonging to the proof-search tree is added as constraints to the proof search.

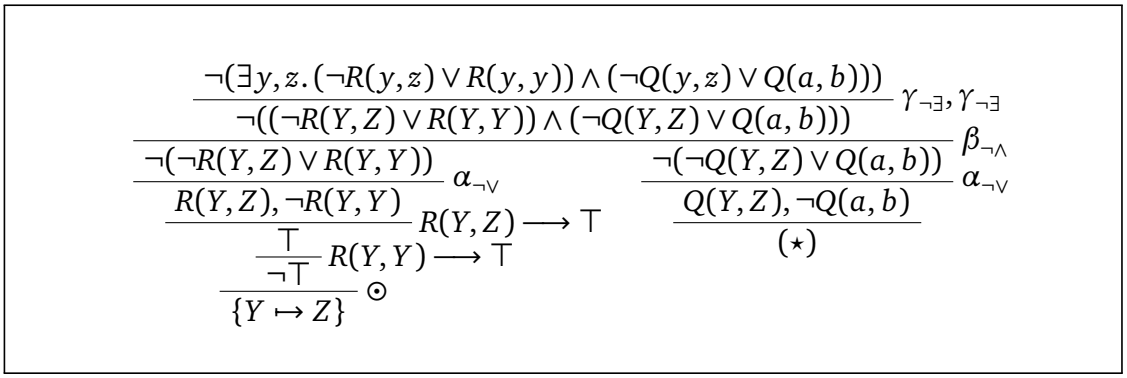


Figure 5.5: Proof-search tree with additional constraints due to atomic rewriting.

Example 5.3: Interaction between the Rewriting Relation and the Proof-Search Procedure

Let us consider the following rewrite rules and incoming formulas:

- $P(X,X) \longrightarrow_{\mathcal{R}} \top$ and $P(a,a)$. Then, $P(a,a) \longrightarrow_{\mathcal{R}} \top$ thanks to $\sigma = \{X \mapsto a\}$, $\top$ is added to the proof search and $P(a,a)$ is removed without additional constraint.
- $P(a,a) \longrightarrow_{\mathcal{R}} \top$ and $P(X,X)$. Then, nothing happens, as there is no substitution σ such that $P(X,X) = \sigma(P(a,a))$.
- $P(X,X) \longrightarrow_{\mathcal{R}} \top$ and $P(Y,Z)$. Then, $P(Y,Z) \longrightarrow_{\mathcal{R}} \top$ thanks to $\sigma = \{Y \mapsto X, Z \mapsto X\}$, $\top$ is added to the proof search and $P(a,a)$ is removed with the additional constraint $\{Y \mapsto Z\}$.
- $P(X,X) \longrightarrow_{\mathcal{R}} \top$ and $P(a,a) \wedge \top$. Then, $P(a,a) \wedge \top \longrightarrow_{\mathcal{R}} \top \wedge \top$ thanks to $\sigma = \{X \mapsto a\}$, $\top \wedge \top$ is added to the proof search and $P(a,a) \wedge \top$ is removed without additional constraint.

However, constraints yields by the addition of rules of the form $P \longrightarrow \top$ into the proof-search procedure can become challenging in the context of a free-variable tableau-based theorem prover, as illustrated in Figure 5.5. In this example, the axiom $\forall x.\, R(x,x)$ has been transformed into the rewrite rule $R(X,X) \longrightarrow_{\mathcal{R}} \top$ using the previous heuristics. Thus, in the left branch, two rewrite rules can be applied: one to $R(Y,Z)$, yielding the constraint $\{Y \mapsto Z\}$, and one to $R(Y,Y)$, leading to a closure. In the case of a proof search without deduction modulo theory, a contradiction could have been found between $\forall x.\, R(x,x)$ and $R(Y,Y)$, which disappeared from the initial problem due to the creation of the rewrite system. As a result, the proof search has to choose a predicate to be rewritten. If $R(Y,Y)$ is chosen first, then the branch can be closed and a proof s found as in the original situation. If $R(Y,Z)$ is chosen first, it generates $\top$ and the additional constraint $\{Y \mapsto Z\}$. This does not prevent the closure in the left branch, since the rewrite rule can still be applied on $R(Y,Y)$, but impacts the right one. Indeed, the node labeled $(\star)$ will attempt to close the branch by substituting Y by a and Z by b, before checking the constraint from its

sibling branch. Consequently, the branch cannot be immediately closed anymore. This second situation is illustrated on Figure 5.5.

As can be seen in this example, the constraints returned by the computation rule may not be useful when closing a branch, especially when an atomic formula has been rewritten into $\top$. As such, we chose not to implement this first heuristics, and to strictly stick to the second. It thus allows the rewriting steps to be fully enclosed inside the rewrite system, and to keep the interactions with the proof-search procedure to a minimum. This problem also highlights the need for a fairness management over the rewriting mechanism, as well as the loss of completeness induced by the application of deduction modulo theory with free-variable tableaux. Completeness can be restored by the use of *narrowing*, i.e., the use of *unification* rather than *pattern matching* in Definition 5.6. However, completeness is not necessarily desirable when it comes to practical performances [73].

The previously described behaviors are the key to ensuring the in-practice termination of a heuristics-based rewrite system. As such, the following rewriting algorithm is proposed:

1. When a literal (an atomic formula or a negated atomic formula) is generated, check for closure.

2. If no closure is found, apply one step of proposition rewriting.

3. If the literal can be rewritten, continue the proof search with the updated formulas.

With the chosen heuristics, if a literal is rewritten, then the resulting formula is not a literal. As such, only one step of rewriting can be applied, thus leading to the obvious termination of the rewrite strategy. The whole process is formalized in Figure 5.6 where $\equiv_{\mathcal{R}}$ is the symmetric closure of $\longrightarrow^*_{\mathcal{R}}$. The chosen formalization follows closely the rewriting algorithm and the Poincaré principle [18] (which states that computation's traces should not be part of the proofs), thus rewriting silently on a rule application if it yields a literal.

It is still important to note that theoretically non-terminating rewrite systems crafted by the heuristics may still lead to a non-termination of the proof search despite taking the aforementioned precautions, but such a system is rare and, in practice, this algorithm is a good compromise between efficiency and simplicity of implementation.

5.2.2 Useful Variants for a Tableaux Proof-Search Procedure

The goal of deduction modulo theory is to transform as many axioms as possible into rewrite rules, to reduce the proof-search space needed by trading proof steps for computation steps. Given the effectiveness and practicality of transforming equivalences such as $\forall \vec{x}.\ P \Leftrightarrow A$, it is reasonable to question the feasibility of translating axioms of different forms. With the basic principles of deduction modulo theory established, this section delves into some optimizations for the deduction modulo theory reasoner: the *polarized* deduction modulo theory and the *preskolemization*.

$$\frac{\bot}{\odot}\,\odot \qquad \frac{\neg\top}{\odot}\,\odot \qquad \frac{F, \neg G}{\odot/\sigma}\,\odot_\sigma, \sigma(F) = \sigma(G)$$

(a) Closure rules.

$$\frac{\neg\neg F}{A}\,\alpha_{\neg\neg},\ F \equiv_{\mathcal{R}} A \qquad \frac{F \wedge G}{\begin{array}{c}A\\B\end{array}}\,\alpha_{\wedge}, \begin{array}{l}F \equiv_{\mathcal{R}} A\\ G \equiv_{\mathcal{R}} B\end{array}$$

$$\frac{\neg(F \vee G)}{\begin{array}{c}\neg A\\ \neg B\end{array}}\,\alpha_{\neg\vee}, \begin{array}{l}F \equiv_{\mathcal{R}} A\\ G \equiv_{\mathcal{R}} B\end{array} \qquad \frac{\neg(F \Rightarrow G)}{\begin{array}{c}A\\ \neg B\end{array}}\,\alpha_{\neg\Rightarrow}, \begin{array}{l}F \equiv_{\mathcal{R}} A\\ G \equiv_{\mathcal{R}} B\end{array}$$

(b) α-rules.

$$\frac{F \vee G}{A \quad B}\,\beta_{\vee},\ F \equiv_{\mathcal{R}} A \qquad \frac{\neg(F \wedge G)}{\neg A \quad \neg B}\,\beta_{\neg\wedge}, \begin{array}{l}F \equiv_{\mathcal{R}} A\\ G \equiv_{\mathcal{R}} B\end{array}$$

$$\frac{F \Rightarrow G}{\neg A \quad B}\,\beta_{\Rightarrow}, \begin{array}{l}F \equiv_{\mathcal{R}} A\\ G \equiv_{\mathcal{R}} B\end{array} \qquad \frac{F \Leftrightarrow G}{\begin{array}{cc}A & \neg A\\ B & \neg B\end{array}}\,\beta_{\Leftrightarrow}, \begin{array}{l}F \equiv_{\mathcal{R}} A\\ G \equiv_{\mathcal{R}} B\end{array}$$

$$\frac{\neg(F \Leftrightarrow G)}{\begin{array}{cc}A & \neg A\\ \neg B & \neg B\end{array}}\,\beta_{\neg\Leftrightarrow}, \begin{array}{l}F \equiv_{\mathcal{R}} A\\ G \equiv_{\mathcal{R}} B\end{array}$$

(c) β-rules.

$$\frac{\exists x.\, F}{G}\,\delta_{\exists},\ F[x \mapsto sko\,(y_1, \ldots, y_n)] \equiv_{\mathcal{R}} G \qquad \frac{\neg\forall x.\, F}{\neg G}\,\delta_{\neg\forall},\ F[x \mapsto sko\,(y_1, \ldots, y_n)] \equiv_{\mathcal{R}} G$$

(d) δ-rules.

$$\frac{\forall x.\, F}{G}\,\gamma_{\forall},\ F[x \mapsto X] \equiv_{\mathcal{R}} G \qquad \frac{\neg\exists x.\, F}{\neg G}\,\gamma_{\neg\exists},\ F[x \mapsto X] \equiv_{\mathcal{R}} G$$

(e) γ-rules.

Figure 5.6: Free-variable tableau rules modulo in a rewrite system $\mathcal{R}$.

Polarized Deduction Modulo Theory By remarking that $P \Leftrightarrow A$ is actually a shortcut for $(P \Rightarrow A) \wedge (A \Rightarrow P)$, and that $P \longrightarrow A$ is the rewrite rules yielded by the original formula, then axioms of the form $\forall \vec{x}.\ P \Rightarrow A$ appear to be promising contenders for an effective translation into rewrite rules. This idea is behond the concept of *polarized deduction modulo theory* [113].

The actual transformation follows closely this intuition, even though some care is needed to retain the soundness of the proof. To formalize this, let us first define a concept that will be subsequently needed: the *polarity* of a predicate.

Definition 5.8: Polarity Function

Let A be a formula and B be a subformula of A. The polarity function Pol takes these two formulas as arguments and returns an integer in $\{0,1\}$ such that $\text{Pol}(B,A) = 0$ if B has a *negative polarity* in A and 1 otherwise. It is defined by induction on A.

- If $A = B$ then $\text{Pol}(B,A) = 1$.
- If $A = \neg F$, then $\text{Pol}(B,A) = 1 - \text{Pol}(B,F)$.
- If $A = F_1 \vee F_2$ or $A = F_1 \wedge F_2$ then if B is a subformula of F_1, $\text{Pol}(B,A) = \text{Pol}(B,F_1)$. Otherwise, $\text{Pol}(B,A) = \text{Pol}(B,F_2)$.
- If $A = F_1 \Rightarrow F_2$ and B is a subformula of F_1 then $\text{Pol}(B,A) = 1 - \text{Pol}(B,F_1)$. Otherwise, $\text{Pol}(B,A) = \text{Pol}(B,F_2)$.
- If $A = \forall x\ F$ or $A = \exists x\ F$ then $\text{Pol}(B,A) = \text{Pol}(B,F)$.

The case of a predicate is clearly a subcase of the first case as if A is an atom, then the only subformula of A is itself.

In short, a formula B has a positive occurrence in a formula A if it appears under a negation an even number of times. Otherwise, it is said to have a negative occurrence.

Example 5.4: Polarity Function

Let us consider the formula $(\neg A) \Rightarrow B$. Then, the polarities of its literals are the following:

- B has a positive occurrence:

$$\begin{aligned}\text{Pol}(B,(\neg A) \Rightarrow B) &= \text{Pol}(B,B)\\ &= 1\end{aligned}$$

- A has a positive occurrence:

$$\begin{aligned}\text{Pol}(A,(\neg A) \Rightarrow B) &= 1 - \text{Pol}(A,\neg A)\\ &= 1 - (1 - \text{Pol}(A,A))\\ &= 1 - (1-1)\\ &= 1\end{aligned}$$

- $\neg A$ has a negative occurrence:

$$\begin{aligned}\text{Pol}(\neg A,(\neg A) \Rightarrow B) &= 1 - \text{Pol}(\neg A,\neg A)\\ &= 1 - 1\\ &= 0\end{aligned}$$

With these new candidates for rewriting rules, an examination of the interaction with the actual rewrite mechanism, based on equivalences, is needed. Thanks to the notion of polarity, new rewrite rules can be deduced from implications (the detailed procedure will be outlined subsequently). Moreover, structurally speaking, equivalence can be seen as a combination of two implications, and thus two rules can be extracted for the initial formula.

In terms of semantics, the first subformula implies that if P holds true, so does A. Consequently, if P appears during the proof search, it can be rewritten into A. However, with only $P \Rightarrow A$, this is not possible for $\neg P$ as it does not imply $\neg P \Rightarrow \neg A$. Nevertheless, the initial formula $P \Leftrightarrow A$ includes $A \Rightarrow P$, which is, by contraposition, $\neg P \Rightarrow \neg A$. Thus, $\neg P$ can be soundly rewritten into $\neg A$.

This semantics is hidden in the rewrite rule $P \longrightarrow A$, which might create the impression that only P is subject to rewriting. Nevertheless, the underlying mechanism is somewhat intricate, and even though polarity does not directly impact the rewriting of equivalence, it is crucial to elucidate this aspect to effectively manage implication rewriting.

As such, an axiom with an equivalence as its root connective naturally yields a positive and a negative rewrite rule. Thus, the previously defined rewrite system $\mathcal{R}$ can actually be split into two disjoint rewrite systems $\mathcal{R}^+$ and $\mathcal{R}^-$, where $\mathcal{R} = \mathcal{R}^+ \cup \mathcal{R}^-$. In this system, an axiom of the form $P \Rightarrow Q$ pertains to $\mathcal{R}^+$ if and only if $\text{Pol}(P, P \Rightarrow Q) = 1$, and to $\mathcal{R}^-$ otherwise. These new systems are called *polarized* rewrite systems, and a *polarized* rewrite relation can be defined analogously as it had been for the standard rewrite system.

Definition 5.9: Polarized Rewrite System

Let $\mathcal{R} = \mathcal{R}^+ \cup \mathcal{R}^-$ be a rewrite system and A, B be formulas. A rewrites *positively* (resp. *negatively*) in B and denoted $A \longrightarrow_+ B$ (resp. $A \longrightarrow_- B$) iff there exists $(P, Q) \in \mathcal{R}^+$ (resp. $\mathcal{R}^-$) such that $\sigma(A) = P$ and $\sigma(B) = Q$.

It can be remarked that $A \longrightarrow_- B$ if and only if $\neg A \longrightarrow_+ \neg B$, i.e., polarities are exchanged. As before, let $\longrightarrow_+^*$ and $\longrightarrow_-^*$ be the reflexive and transitive closure of their respective relations. As this formalization is at least as powerful as the standard deduction modulo theory rewrite system, it is clear that in general, neither confluency nor termination is ensured.

Nevertheless, heuristically turning axioms of this particular type into rewrite rules offers a great increase in terms of number of rewrite rules computed over standard deduction modulo theory:

$$\begin{array}{rcl} \forall \vec{x}.\, P \Rightarrow A & \text{becomes} & P \longrightarrow_- A \\ \forall \vec{x}.\, A \Rightarrow P & \text{becomes} & P \longrightarrow_+ A \\ \forall \vec{x}.\, \neg P \Rightarrow A & \text{becomes} & P \longrightarrow_+ A \\ \forall \vec{x}.\, A \Rightarrow \neg P & \text{becomes} & P \longrightarrow_- A \end{array}$$

where P is still a non-atomic and A anything other than a literal. It can be noted that in this particular case, allowing A to be a literal in the heuristic does not immediately lead

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x.\, \neg P(x) \wedge (Q(x) \vee Q(x))}{\neg P(X) \wedge (Q(X) \vee Q(X))}\,\gamma_\forall}{\neg P(X), Q(X) \vee Q(X)}\,\alpha_\wedge}{Q(X) \longrightarrow \exists y\, P(y)}\,\beta_\vee}{\dfrac{Q(X)}{\exists y\, P(y)}\,\delta_\exists}}{P(c_1)}}{\{X \mapsto c_1\}}\,\odot \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x.\, \neg P(x) \wedge (Q(x) \vee Q(x))}{\neg P(X) \wedge (Q(X) \vee Q(X))}\,\gamma_\forall}{\neg P(X), Q(X) \vee Q(X)}\,\alpha_\wedge}{Q(X) \longrightarrow \exists y\, P(y)}\,\beta_\vee}{\dfrac{Q(X)}{\exists y\, P(y)}\,\delta_\exists}}{P(c_2)}}{\neg P(X') \wedge (Q(X') \vee Q(X'))}\,\gamma_\forall}{\neg P(X'), Q(X') \vee Q(X')}\,\alpha_\wedge}{\{X' \mapsto c_2\}}\,\odot$$

(a) Lengthy proof in standard deduction modulo.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x.\, \neg P(x) \wedge (Q(x) \vee Q(x))}{\neg P(X) \wedge (Q(X) \vee Q(X))}\,\gamma_\forall}{\neg P(X), Q(X) \vee Q(X)}\,\alpha_\wedge}{Q(X) \longrightarrow P(c)}\,\beta_\vee}{\dfrac{Q(X)}{P(c)}}}{\{X \mapsto c\}}\,\odot \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x.\, \neg P(x) \wedge (Q(x) \vee Q(x))}{\neg P(X) \wedge (Q(X) \vee Q(X))}\,\gamma_\forall}{\neg P(X), Q(X) \vee Q(X)}\,\alpha_\wedge}{Q(X) \longrightarrow P(c)}\,\beta_\vee}{\dfrac{Q(X)}{P(c)}}}{\{X \mapsto c\}}\,\odot$$

(b) Shorter proof in polarized deduction modulo by preskolemizing.

Figure 5.7: Improvement of proof search by preprocessing formulas.

to non-terminating systems, but it should be avoided as the proof-search procedure will often get lost trying to compute these rules over searching for a proof. In a nutshell, polarizing the heuristic allows more axioms to correspond to rewrite rules, thus improving the overall efficiency of the proof-search procedure as some space is again relieved of its load. Furthermore, in practice, as tableau rules keep the syntactic integrity of the formulas processed, it suffices to pick the system corresponding to the polarity of the literal when trying to rewrite, allowing a straightforward implementation of the method.

Preskolemization However, the real gain of polarization does not solely lie in the removal of some axioms from the proof-search space, but comes from the ability to *preprocess* some specific rewrite rules. It is widely known that in every logic, the complexity of searching for a proof is closely related to the presence of quantifiers in the formula to (dis)prove. Indeed, any theory which admits the elimination of quantifiers, such as the algebraically closed fields, is decidable. In first-order logic, research towards eliminating quantifiers has led to the Skolem-Herbrand's theorem [147, 218], which states that any formula can be translated into either (i) a solely universally-quantified formula or (ii) a uniquely existentially-quantified formula. Both methods preserve some properties of the initial formula, such as the satisfiability for (i) and the validity for (ii). In proving by refutation, the equisatisfiability of (i) is needed, and thus

existentially quantified formulas are *Skolemized* during the proof search. However, the native process of Skolemization of an existentially bound variable has to yield a *fresh* function symbol, which differs even when the same formula is Skolemized multiple times. As such, it can prevent early closure of a branch when Skolemizing twice the same formula in two different branches, as can be seen in Figure 5.7a. In this example, the formula $\forall x.\, Q(x) \Rightarrow \exists y\ P(y)$ was turned into the rewrite rule $Q(X) \longrightarrow \exists y P(y)$.

However, as polarizing a rewrite system allows us to *know* how to rewrite both for positive and negative cases, it also enables us to *know* the polarity of the rewritten formula. Thus, such a step could be avoided by *preskolemizing* positive occurrences of existentially-quantified formulas and negative occurrences of universally-quantified formulas, yielding the shorter proof of Figure 5.7b. In this example, the formula $\forall x.\, Q(x) \Rightarrow \exists y\ P(y)$, the occurrence of $\exists y P(y)$ was preskolemized, resulting in $P(c)$ being directly integrated under this form. This leads to the rewrite rule $Q(X) \longrightarrow P(c)$ and a shorter proof. This example makes use of inner Skolemization (i.e., the Skolemization creates a function symbol that takes as parameters the free variables within the predicate) although it also holds for other Skolemization strategies.

Applying this kind of preprocessing over the heuristically-generated polarized rewrite system for the set theory's axiom of inclusion $\forall a, b.\, a \subseteq b \Leftrightarrow \forall x.\, x \in a \Rightarrow x \in b$ will thus generate the following rewrite rules, shortening the proof 5.4b by one step.

$$A \subseteq B \longrightarrow_{+} \forall x.\, x \in A \Rightarrow x \in B$$
$$A \subseteq B \longrightarrow_{-} f(A,B) \in A \Rightarrow f(A,B) \in B$$

Thus, even though uniform Skolem symbols could be attained by smarter Skolemization strategies, such as the δ^{++} rule [36] which natively handles the yielding of the same symbol of Skolem for every formula in the same α-equivalence class, preprocessing formulas by preskolemizing still offers a slight advantage over standard deduction modulo theory, which scales up pretty well on problems generating an important proof-search space.

5.2.3 Key points of the Interaction with the Proof-Search Procedure

As highlighted multiple times in Subsection 5.2.1, the key to building an efficient rewrite system in an automated theorem-proving procedure lies in how the former interacts with the latter. This subsection discusses some pathological problems arising when setting up this interaction.

Loss of Cut-Free Completeness While deduction modulo theory offers numerous advantages, it can potentially result in a loss of completeness, moreover if the calculus does not incorporate the *cut* rule. The cut rule, originally designed in Gentzen's sequent calculus, can be adapted to the tableau format [220] as follows, where F represents any first-order formula:

$$\frac{}{F \qquad \neg F}\ cut$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall x.\, P(x) \Leftrightarrow (P(x) \Rightarrow Q(x)), \forall y\ \neg Q(y)}{(P(X) \Leftrightarrow (P(X) \Rightarrow Q(X))), \neg Q(Y)}\,\gamma_\forall \times 2}{\dfrac{P(X), P(X) \Rightarrow Q(X)}{\dfrac{\neg P(X)}{\odot}\,\odot \quad \dfrac{Q(X)}{\{X \mapsto Y\}}\,\odot}\,\beta_\Rightarrow} \quad \dfrac{\neg P(X), \neg(P(X) \Rightarrow Q(X))}{\dfrac{P(X), \neg Q(X)}{\odot}\,\odot}\,\alpha_{\neg\Rightarrow}}\,\beta_\Leftrightarrow$$

(a) Tableau proof that $\forall x.\, P(x) \Leftrightarrow (P(x) \Rightarrow Q(x)), \forall y\ \neg Q(y)$.

$$\dfrac{\dfrac{\dfrac{\forall y\ \neg Q(y)}{\neg Q(Y)}\,\gamma_\forall}{\neg Q(Y')}\,\gamma_\forall}{\cdots}\,\gamma_\forall$$

(b) Tableau proof with deduction modulo theory that $\forall x.\, P(x) \Leftrightarrow (P(x) \Rightarrow Q(x)), \forall y\ \neg Q(y)$.

$$\dfrac{\dfrac{\forall y\ \neg Q(y)}{\neg Q(Y)}\,\gamma_\forall}{\dfrac{\forall x.\, P(x) \Leftrightarrow (P(x) \Rightarrow Q(x))}{\text{Similar to 5.8a}}\,\odot \quad \dfrac{\dfrac{\neg(\forall x.\, P(x) \Leftrightarrow (P(x) \Rightarrow Q(x)))}{\neg(P(c) \Leftrightarrow (P(c) \Rightarrow Q(c)))}\,\delta_{\neg\forall}}{\dfrac{\dfrac{\neg P(c), P(c) \Rightarrow Q(c)}{\dfrac{\neg P(c)}{\dfrac{\neg(P(c) \Rightarrow Q(c))}{P(c), \neg Q(c)}\,\odot}\,\alpha_{\neg\Rightarrow}}\,\star \quad \dfrac{Q(c)}{\odot}\,\odot}\,\beta_\Rightarrow \quad \dfrac{P(c), \neg(P(c) \Rightarrow Q(c))}{\dfrac{P(c), \neg Q(c)}{\dfrac{P(c) \Rightarrow Q(c)}{\dfrac{\neg P(c)}{\odot}\,\odot \quad \dfrac{Q(c)}{\odot}\,\odot}\,\beta_\Rightarrow}\,\star}\,\alpha_{\neg\Rightarrow}}\,\beta_\Leftrightarrow}\,cut$$

With $\star = P(X) \longrightarrow (P(X) \Rightarrow Q(X))$.

(c) Tableau proof with deduction modulo theory and the *cut* rule that $\forall x.\, P(x) \Leftrightarrow (P(x) \Rightarrow Q(x)), \forall y\ \neg Q(y)$.

Figure 5.8: Loss of cut-free completeness due to the use of deduction modulo theory.

Unfortunately, as observed in Figure 5.8, inspired by [112], deduction modulo theory does not preserve completeness in the cut-free case. In this example, the original proof with the tableau method is presented in Figure 5.8a, whereas in Figure 5.8b and Figure 5.8c, the axiom $\forall x.\, P(x) \Leftrightarrow (P(x) \Rightarrow Q(x))$ was transformed into the rewrite rule $P(X) \longrightarrow (P(X) \Rightarrow Q(X))$. In the first case, as only $\forall y\ \neg Q(y)$ remains in the tableau, the only available rule is the application of a γ-rule that generates $\neg Q(Y)$, and no proof can be found. However, by adding the cut rule in Figure 5.8c, we can introduce two new elements, $P(X) \Leftrightarrow (P(X) \Rightarrow Q(X))$ and its negation, and recover the proof.

However, for a given theory, it is undecidable to know whether or not the cut rule is needed [74]. Moreover, since this rule can theoretically introduce any axiom into the proof search and be triggered at any time, it is not well-suited for automated deduction, although some restrictions can be applied [95].

Non-Confluency As said in Subsection 5.2.1, in general, a heuristically-generated rewrite system is not confluent, and also non-terminating. Thus, when implementing deduction modulo theory, losing completeness due to non-confluent systems happens easily. Indeed, such systems often appear in most theories. For example, let us take the usual naive set theory, where equality between two sets (also known as the axiom of extensionality) can be expressed in the following ways:

$$\forall a, b.\, a = b \Leftrightarrow a \subseteq b \wedge b \subseteq a$$
$$\forall a, b.\, a = b \Leftrightarrow (\forall x.\, x \in a \Leftrightarrow x \in b)$$

The system for turning axioms into rewrite rules presented in the previous subsections yields the following rewrite system for these axioms:

$$A = B \longrightarrow A \subseteq B \wedge B \subseteq A$$
$$A = B \longrightarrow \forall x.\, x \in a \Leftrightarrow x \in b$$

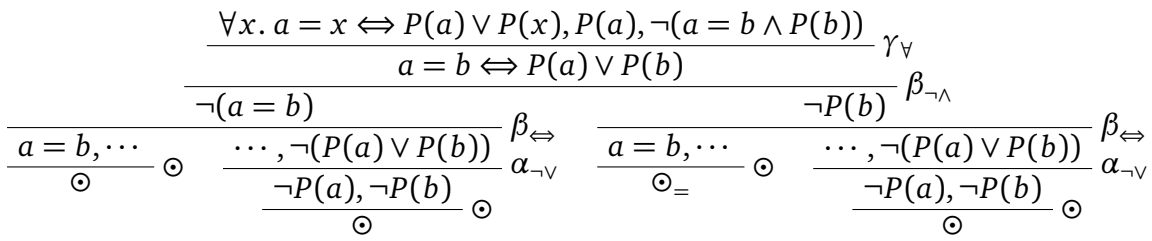
There are thus two different possibilities available when trying to rewrite atoms expressing the equality of two sets, and the rewrite mechanism has to choose how to rewrite the incoming formula. In this case, naive solutions by orienting the proof search do not suffice to guarantee the completeness of the procedure and to ensure it, rewriting *has to* become a backtrack point in the procedure.

Constraints-Yielding Rewriting with Narrowing The use of *narrowing* allows us to preserve the completeness of the proof search. Those interactions have also been briefly considered in Subsection 5.2.1 when rewriting atomic axioms to true propositions. It should be clear that in general, rewriting an atom will lead to additional constraints to be included in the subsequent proof-search steps. This is especially true in case of narrowing instead of simple rewriting as defined in Definition 5.6. As narrowing allows us to apply a substitution on both sides — left-hand side part of the rewrite rule and the incoming atom — it necessarily leads to a larger amount of constraints. For example, it is necessary to be extra-careful when rewriting an atom $P(t_1, \ldots, t_n)$ where for some i, t_i is a function $f(t'_1, \ldots, t'_k)$. This principle can be illustrated fairly simply by using the following rewrite rule:

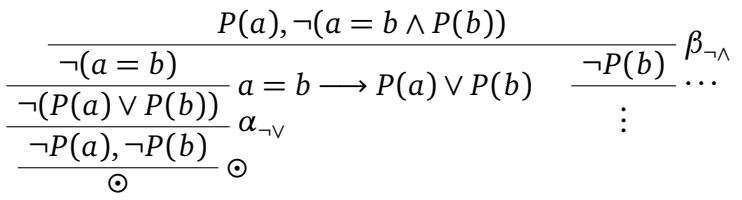
$$P(X, f(Y)) \longrightarrow Q(X) \wedge R(X, Y)$$

Then, the incoming atom $P(a, Z)$ with Z being some free variable instantiated by the proof search is rewritten into $Q(a) \wedge R(a, Y)$ with the constraint $\{Z \mapsto f(Y)\}$. It thus introduces a free variable in the proof search that is non-native to it. It is thus necessary to pay specific attention to these free variables in order to keep the consistency of the proof search. Moreover, narrowing may be costly and might be only sparingly implemented, for example on certain predicates or at certain step of the proof search.

Interactions with Equality As equality and deduction modulo theory is intrinsically *distinct* modules (i.e., no equational rewriting is done), they may clash if left alone together. Indeed, rewriting systems integrate seamlessly into proof-search procedures



(a) Standard tableau proof with equality.



(b) Open tableau-modulo due to loss of information.

Figure 5.9: Loss of completeness on equational axiom rewrite.

as they also reason syntactically over formulas, while equality reasoners care more about the semantics involved in the relevant axioms. As such, rewriting those axioms leads to a semantic loss of information and thus quickly yields an incomplete proof-search procedure, as Figure 5.9 shows. Actually, this choice of not rewriting relevant axioms should be kept for any theory integrated into the proof-search procedure needing some kind of semantics reasoning.

Embodiment into the Proof-Search Procedure In the deduction modulo theory, when an atomic formula is generated, an attempt is made to unify it with available rewrite rules. If a match is found, the formula is rewritten, and the proof search continues. The original formula, along with other possible rewrite options, is retained as backtracking points. If none of them lead to a solution, the original formula is tried in turn. The deduction modulo theory mechanism necessitates a shift in the backtracking point from substitution to rewrite. Consequently, when a branch reaches its limit, backtracking is performed at the last backtracking point (either rewrite or substitution). A history of the backtracking point is also maintained.

Figure 5.10 illustrates two distinct types of backtracking points. Let us consider the rewrite rule $F \longrightarrow G$. The first backtracking point appears in n_3, when the substitution found by n_1 is applied to n_2, generating a twin. Starting now, if a branch reaches its limit, it should backtrack on σ', the substitution found by n_2. However, n_3 continues its proof search, generating two children. One of them, n_5, generates F, which is rewritten into G in a new node n_6, thanks to the rewrite rule. From this moment on, if the subsequent branch of n_6 is open, the backtrack should be performed on F rather than σ'.

Some heuristics can be envisioned in order to reduce the backtracking factor.

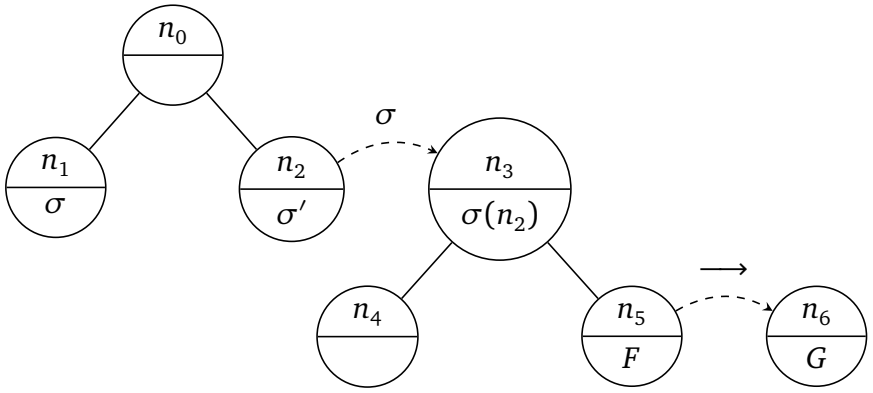


Figure 5.10: Double backtracking points: substitutions and rewrite rules.

For example, at the current time, a rule that rewrites a proposition into $\bot$ is always preferred to others. In addition, we can imagine adding a ranking over rewrite rules for the same axiom by carefully studying the rewrite system or limiting the backtrack to the rewrite rules that add constraints to the proof search.

5.3 Conclusion

This chapter has presented a concrete example of a specific theory, equality, along with a method to generalize theory handling using deduction modulo theory.

Given its widespread use, intuitive nature, and expressiveness, equality reasoning has garnered global interest, leading to the development of various algorithms over the years to manage it accurately. However, it remains one of the most challenging theories to handle, influencing all branches during its application and having a huge potential search area. Due to its unique characteristics and prevalence in problems, equality reasoning can either be directly embodied into the proof-search process or implemented externally as a background reasoner [33, 124]. Since this is a fundamental but hard-to-handle theory, research into efficient equality reasoning within tableau-based frameworks continues to evolve.

In a broader sense, we have presented deduction modulo theory, a general-purpose technique tailored to reason within theories, its variants, and its integration in a free-variable tableau-based automated theorem prover. This technique allies simplicity and efficiency to yield shorter proofs in axiomatized theories. While performances cannot challenge those of dedicated background reasoners, the advantage of handling any theory is significant, striking a balance between efficiency and extensibility. Moreover, these techniques are not mutually exclusive: a system based on deduction modulo theory can coexist alongside specific background reasoners. As complex theories continue to emerge, provers have been prompted to evolve in response.

The management of theories in tableau-based theorem prover is not uniform. Whereas some of them can act in a very independent way, others require a high degree of communication and fit well with concurrency. We have studied the implementation of the two previous background reasoners into a concurrent proof-search procedure, highlighting the points that can be parallelized and the key interactions

with the proof search.

Ultimately, theory reasoning has evolved into an essential component of any automated theorem prover, and this trend will continue as problems grow larger and more specialized. The use of background reasoners and their collaboration with foreground components plays a crucial role in the development of such tools, requiring careful attention to their interaction with proof-search procedures.

Chapter 6
Goéland: A Concurrent Tableau-Based Theorem Prover

Contents

This chapter describes the implementation of a concurrent automated theorem prover: Goéland [80]. It is available on GITHUB at the following link: https://github.com/GoelandProver/Goeland. This section highlights the main points of implementation and details its typing mechanism.

The implementation of the core of the prover, i.e., the procedure introduced in Chapter 3, is presented in Section 6.1. This section places a particular emphasis on key mechanisms and data structures, with the aim of facilitating its reproducibility. The management of typed problems is exposed in Section 6.2. This extension broaden the scope of Goéland, allowing it to tackle a wider range of problems.

6.1 Implementation of the Concurrent Proof-Search Procedure

As the name suggests, Goéland is developed in the Go programming language. Go is notable for its support of concurrency and parallelism, which is primarily facilitated through lightweight execution threads known as *goroutines* [231]. Goroutines are executed according to a so-called hybrid threading (or M:N) model: M goroutines are executed over N effective threads and scheduling is managed by both the Go runtime and the operating system. This threading model allows the execution of a large number of goroutines with a reasonable consumption of system resources.

Formulas:

- $P(g(a,x),c)$
- $P(g(x,b),x)$
- $P(g(a,b),a)$
- $P(g(x,c),b)$
- $P(x,y)$
- $P(x,z)$

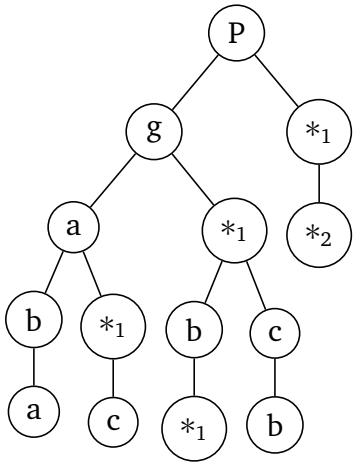


Figure 6.1: A set of atoms and the corresponding discrimination tree.

Goroutines use channels to exchange messages so that the implementation is close to the presentation in Section 3.2. This being said, this section delves into the core of the implementation of the prover, provides technical insights, and addresses the management of various extensions previously mentioned.

6.1.1 Key Mechanisms and Data-Structure

In Goéland, the implementation of basic elements such as terms, formulas, and substitutions is relatively straightforward and does not warrant specific explanations. However, some other aspects requiring a more detailed treatment are explained below.

Unification The performance of unification relies heavily on the chosen term indexing method and the unification algorithm itself. Consequently, selecting the appropriate approach is a pivotal decision in designing an automated theorem prover. Drawing from [185], we opted to implement *code trees* as described in [233], with extensions tailored to unification. Code trees can be thought of as a compiled version of discrimination trees, with both structures exclusively storing atoms.

Code trees have been originally designed to perform *subsumption*, but we have adapted the mechanism to be able to deal with *unification*. In the subsumption version, the goal is to determine whether one atom, say P_2, subsumes another, P_1, which means finding a substitution σ such that $\sigma(P_1) = P_2$. In other words, every element in $\sigma(P_1)$ belongs to P_2. Conversely, our implementation seeks to find a unifier between P_1 and P_2.

To get back to the original idea, discrimination trees treat terms as strings, sharing common prefixes. Variables, represented by stars ($*$), are indexed (i.e., $*_1$, $*_2$) to distinguish them. A tree can be empty, a node, or a leaf. A node consists of a tree element, such as a constant, function symbol along with its arity, or a star variable. Leaves, which are a special type of node, additionally contain all the formulas corresponding to the branch. An example of a set of formulas and the corresponding discrimination tree can be seen Figure 6.1.

Code trees are based on an abstract subsumption machine, which converts discrimination trees into machine instructions. We extended this subsumption machine by incorporating additional operations, such as the beginning and the end of a function. To achieve unification, each atom is translated into a sequence of instructions and "applied" to the code trees. At the end of this process, if the sequence has been successfully applied, the unification process returns a set of substitutions, or an empty one if no solution has been found.

This machine is based on two arrays and a set of instructions. Put together, a code tree is a sequence of instructions, describing an atom. The machine works with a cursor q, indicating the current term. Two arrays follow the cursor and are updated all along the unification process: *post*, which records the position of the next element in the function or predicate, and *subst*, which keeps a trace of the substitution to perform at the end of the sequence (i.e., the variable in the code tree are associated to terms in the candidate atom). The original instructions are the following:

- `Initialize`: set q to the initial position.
- `Check P`: check that the term at position q is P.
- `Down`: go down the current position (i.e., from a term to its arguments).
- `Right`: go to the fight (i.e., from an argument in a term to its next argument).
- `Push n`: push the position to the right off the n^{th} position in *post*.
- `Pop n`: set q to the n^{th} position in *post*.
- `Put n`: put the current position on the n^{th} position in *subst* (as the substitution for the n^{th} variable).
- `Compare n m`: compare terms at positions m and n in *subst* (both positions correspond to the substitution for the same variable).
- `Success`: exit with success.
- `Failure`: exit with failure.

To be able to perform unification, two instructions have been added, `Begin` and `End`. These instructions allow us to delimit a block, such as the argument of a function. Moreover, three instructions were deleted compared to the original subsumption machine: `Initialise` (replaced by `Begin`), `Success` and `Failure` (both replaced by `End`). Moreover, we add another array *variables* to manage variables within the candidate atom which needs to be associated with a term in the code tree.

Subsumption	Unification
1 Initialize 2 Check P 3 Down 4 Check f 5 Push 0 6 Down 7 Put 0 8 Right 9 Check a 10 Pop 0 11 Put 1 12 Compare 0 1 13 Success	1 Begin 2 Check P 3 Begin 4 Down 5 Check f 6 Begin 7 Push 0 8 Down 9 Put 0 10 Right 11 Check a 12 Pop 0 13 End 14 Put 1 15 Compare 0 1 16 End 17 End

Table 6.1: Translation of $P(f(X,a),X)$ with the subsumption machine and the unification machine

Furthermore, in alignment with the Goéland philosophy, the search for unification has been parallelized. Since the exploration of the branches is independent, the process stops when all branches have been completely explored.

Proof Search Structure and Interaction with Processes In Goéland, each proof-search node corresponds to a *goroutine*. The creation of a new node occurs under two conditions: (i) when a twin is generated (as introduced in Section 1.3.3), typically resulting from the application of a substitution or a rewriting rule, or (ii) when a beta rule is applied, leading to the generation of two nodes. Nodes interact with each other solely through parent-child communication, meaning that a node can exclusively send messages to its parent or its children.

The messages exchanged between nodes encompass substitutions, lists of formulas, or termination commands (referred to as **kill** orders). When a node is terminated, it also terminates any potential child nodes it may have spawned. Furthermore, due to the concurrent nature of Goéland, multiple executions of the system may not necessarily yield identical outputs. This is because substitutions are managed using a *first received, first tried* approach. Consequently, when multiple potential substitutions are available, the final proofs may diverge.

Proof Construction Let us now focus on the mechanisms which lead to the final proof. As mentioned earlier, Goéland launches its proof-search procedure in parallel on each child when applying a β-rule. These children search autonomously until they encounter a contradiction and discover a substitution. From the beginning of the proof

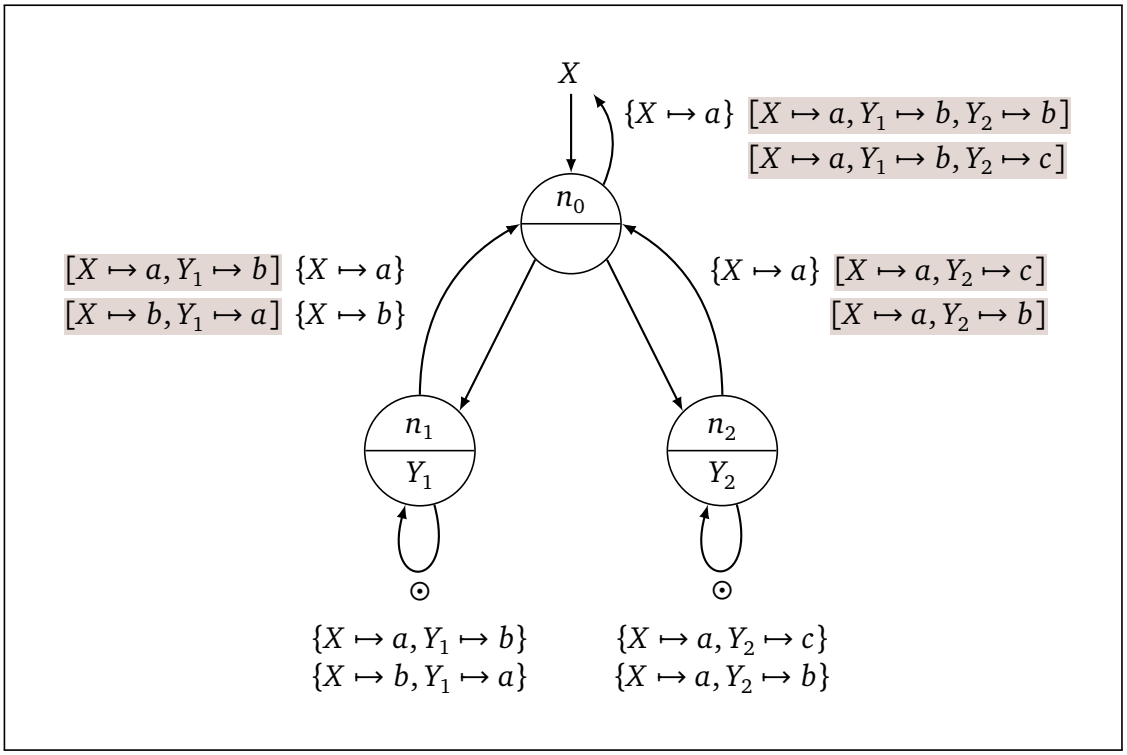


Figure 6.2: Unification management and global unifier in the proof-search procedure.

search, each child maintains a record of the rules they have applied and transmits this history to the parent when a solution is found. This sequence of rules constitutes the *local proof* of a child, which is subsequently merged with those of the other children by the parent node. The union of these local proofs forms the final proof.

As free variables are carried uninstantiated during the proof search, the final proof also contains free variables, which have to be replaced to get a real proof. In order to do this, a substitution is linked to each local proof, to build a *global unifier* and apply it to the final proof. However, by moving upward inside the proof tree, some variables become local, and are thus needed by the global unifier but not by the local agreement mechanism, which only deals with non-local free variables. Thus, to keep track of the local variable, the global unifier is composed of two parts: the *non-local* part, which only contains *non-local* variables, and a set of substitutions resulting from the combination of the answers from the children, including the *local* variables. It ensures the consistency and the correct replacement of free variables at the end of the proof, without involving useless substitution during the agreement mechanism.

Figure 6.2 illustrates this mechanism. The parent n_0 selects the common answer $\{X \mapsto a\}$ and thus, both n_1 and n_2 can be closed. Since free variables are instantiated at the end, we merge all the compatible answer in order to keep track that, for instance, Y_1 need to be mapped to b if $\{X \mapsto a\}$ is chosen. This mechanism offers *at least* one unifier that closes it when the proof-search procedure ends, thus outputting a sound proof.

6.1.2 Variations of the Proof Search

This part presents the implementation of some variations of the proof-search procedure.

Skolemization Three variants of Skolemization have been implemented into Goéland, in order to improve the management of Skolem symbols and free variables dependencies. We implemented the δ, δ^+ and δ^{+^+}-rules, as introduced in Section 2.1.

To achieve this, we need to keep track of free variables present in a branch as well as in a predicate, along with the Skolem symbol associated with a specific γ-formula. Managing the first two cases was relatively straightforward: we added methods to the formula interface to store the internal free variables within it and within the branch. This might initially appear redundant as free variables can usually directly be found inside the formula in question. However, the proof-search of Goéland is *destructive* by nature, which means that once a substitution is found, it is applied to the whole tableau. Consequently, free variables are lost during this process, making the storage of free variables necessary for correctly passing the required arguments during Skolemization. Regarding the last aspect, it merely involves establishing a mapping between γ-formulas and their corresponding Skolem symbols. Furthermore, we have introduced two flags to control the Skolemization mode: `-inner` which activates δ^+-rules and `-preinner` that makes use of δ^{+^+}-rules.

Completeness Mode While completeness is an important characteristic for any theorem prover, it may often slow down the proof-search procedure. In Goéland, completeness is mostly ensured by the prohibition of certain substitutions in cases where the agreement mechanism does not succeed on the first attempt. Empirically, disabling completeness greatly enhances proof-search speed with only a minor loss in problem-solving capability. This is primarily because, in most cases, the first solution found by one of the branches suffices to solve the problem, and completeness delays backtracking to higher nodes. To address this, we no longer consider forbidden substitutions in the procedure described in Section 3.2. This prevents resuming the proof search if no agreement has been found during the first attempt of reconciliation. Instead, we return the message that no solution has been found to the parent node. These substitutions prevent the resumption of proof search if no agreement is reached during the initial reconciliation attempt, allowing faster backtracks, but also a loss of completeness.

Interactive Mode An interactive mode, accessible through the `-interactive` flag, has been implemented into Goéland. In this mode, a console interface allows the user to make decisions at each step regarding which rule to apply, as opposed to relying on the automated proof-search procedure to make the choice. This mode serves various purposes, including comparing Goéland's strategy, ensuring consistent execution (which is not guaranteed due to parallelism), exploring alternative paths, and identifying implementation issues.

6.2 Handling Typed Problems with Polymorphism

In the last decade, we have seen that the multiplicity and diversification of problems have created the need for provers to improve themselves in order to be able to reason with theories. We have seen that some theories, such as equality, need dedicated mechanisms to perform efficiently. This is also the case when we aim to reason in the presence of typed logics.

In typed logics, we aim to provide a more intuitive representation of the world. In these logics, all the terms are *typed*, i.e., they are associated to a *type*. Furthermore, by adding type restrictions on these terms, the proof search avoids irrelevant instantiations and is thus guided in the right direction. A type can be a usual type, such as integers or booleans, or a hand-made type designed for a specific theory. All of them are distinct, with their own properties, and are not interchangeable, i.e., a set of booleans would not accept an integer.

In first-order logic, typing is managed in an *ad-hoc* way, thanks to typing predicates. For instance, for two integers x and y, commutativity can be formally defined as follows:

$$\forall x, y.\, \mathsf{int}(x) \Rightarrow \mathsf{int}(y) \Rightarrow x + y = y + x$$

However, this typing method increases the formulas' size, slowing down the proof search. Moreover, it could be very redundant. For example, in order to define an array of int and an array of boolean, all the properties have to be defined twice, one for each type of array.

To overcome these challenges, research for a more general way to manage typed formulas without overloading the problems has been carried out. Two approaches have emerged: erase the types from the problem or, conversely, natively manage types into a prover. The first one relies on an encoding of typed problems into untyped first-order logic [47], which makes them accessible to a wider range of provers. The second embodies types directly into the terms themselves, adding additional rules to guarantee the *well-typedness* of the formulas.

We have decided to embed many-sorted logic [235] inside Goéland, as to natively type a term. Thus, each term is assigned to a unique type. In particular, this embedding allows Goéland to parse problems with quantification over types. Together with this new notion, the commutativity of addition over integers can now be defined in the following way:

$$\forall x, y : \mathsf{int}.\, x + y = y + x$$

Secondly, we want to be able to generalize properties to all the types. To do this, we want to quantify over types, i.e., to make a predicate applicable to every type. This is called *polymorphism* [138, 207]. This representation allows predicates and functions to accept types as parameters. Thus, the following example holds for every type τ:

$$\forall x : \tau.\, x = x$$

The goal of this section is to present the implementation of a type system that natively manages polymorphism and thus avoids the noise caused by predicate types,

enabling Goéland to reason on typed problems. The typing system was inspired by [73, 78] and implemented following the standard of TPTP library, more precisely the TFF format [48, 228].

6.2.1 Type Definitions and Context

This section aims to define in detail the notion of type and their application on terms by giving them a *type signature*. It also introduces the type system and characterizes formulas regarding whether or not they are type-quantified: *polymorphic* if so, *monomorphic* otherwise.

The first challenge is to build a prover able to reason with types to give a type to terms, i.e., to the variables and functions. To achieve this, the following elements are required:

- A set of type constants: integer, boolean, custom types
- A symbol to type function: $\rightarrow$
- A symbol to type tuples: $\times$

Variables are associated with a type (constant or tuple), whereas functions can take a tuple of types, corresponding to the types of their parameters, and return a type, the one of the function. All of this is summarized into a *type signature*, illustrated in Example 6.1:

Example 6.1: Type Signature

- $(x : \mathsf{int})$: the term x has the type int.
- $(y : \mathsf{bool})$: the term y has the type bool.
- $(f : \mathsf{int} \rightarrow \mathsf{int})$: the function f takes an int as parameter and return an int. This allows us to write $f(x)$.
- $(g : (int \times bool) \rightarrow int)$: the function g takes a pair of $(\mathsf{int}, \mathsf{bool})$ as parameter and returns an int. This allows us to write $g(x, y)$.

Having established the concept of signature for terms, the next step is to incorporate them into formulas. In practice, functions, constants, and predicates are introduced alongside their respective type signatures in a *context*, usually denoted Γ. In contrast, the type of variables is directly given within the formulas themselves, as illustrated in Example 6.2:

Example 6.2: Context and Formulas

Let us consider the following context, composed of types, function symbols, and predicates and their associated type signatures:

- Two types int and $\mathtt{list}$.
- A function $\mathtt{cons}$: $(\mathsf{int} \times \mathtt{list}) \to \mathtt{list}$.
- A function $\mathtt{head}$: $\mathtt{list} \to \mathsf{int}$.
- A predicate $\mathtt{EqualsInt}$: $(\mathsf{int} \times \mathsf{int}) \to \mathsf{bool}$.
- A predicate $\mathtt{EqualsList}$: $(\mathtt{list} \times \mathtt{list}) \to \mathsf{bool}$.

The previous context allows us to build the following formula:

$$\forall(x : \mathsf{int})(\ell, \ell' : \mathtt{list}).\ \mathtt{EqualsList}(\ell', \mathtt{cons}(x, \ell)) \Rightarrow \mathtt{EqualsInt}(\mathtt{head}(\ell'), x)$$

Moreover, the TPTP format defines built-in types, which are recognized and interpreted by the prover [228]. Thus, the following types and operators are natively implemented into Goéland:

- i: type of individuals (variables, constants, ...).
- o: booleans (in particular, the types of the formulas).
- $\mathtt{Type}$: the type of the types.
- Arithmetic types and operators (int, double, $+$, $-$, $=$, ...)

It is important to note that even with the addition of types, reasoning on pure first-order logic is still possible: we only have to give the type i to every term, and o to every formula.

However, we want to achieve greater expressiveness. For example, we would be able to create a list of int or bool, or an equality predicate defined for every type. This constitutes the main idea behind *polymorphism*. A polymorphic type system allows us to quantify over types, i.e., to define properties that work for every type. To achieve this, two new elements are needed: *type constructors* and *type variables*.

Definition 6.1: Type Constructor

A *type constructor* is a n-ary function building a type from types parameters. It takes as an argument a tuple of types to yield a type. A type constructor of arity m is denoted $\mathrm{T} :: m$.

Example 6.3: Type Constructor

The following notation represents a list of a given type Type.

$$\texttt{List} : \text{Type} \rightarrow \text{Type}$$

It could be instantiated by a type, for example, int. It results in a type representing a list of int.

$$\texttt{List}(\text{int}) : \text{Type}$$

Definition 6.2: Type Variables

A *type variable* is a variable representing a type. In contrast to a usual variable, a type variable is instantiated with a type, not by a term. They are often denoted by the symbol α and indexed.

The syntax of polymorphic first-order logic is given in Figure 6.3. It details the structure of a type and a type scheme (that binds type variables) that are used for polymorphic symbols, as well as terms, formulas, and type-quantified formulas. The symbol α is used to denote a type, possibly indexed. Both predicates and functions can be polymorphic and are also parametrized with a type. Consequently, the parameters of functions and predicates are now divided into two parts: one containing the set of types utilized within the function or predicate, and another containing terms employed as parameters themselves. These two parts are separated by a semicolon. Finally, formulas can be quantified over types, with the restriction that those quantifications are universal and occur at the head of the formulas (i.e., in prenex form).

Figure 6.4 extends the notion of context by defining the *local* and *global* contexts, as pairs composed of a symbol and a type. The global context Γ_G contains function symbols, predicate symbols, and constructors, while the local context Γ_L contains terms and type variables and is built through the *typing* process presented in the next section. A typing context Γ is a pair $\Gamma_G; \Gamma_L$.

Finally, the previous grammar allows us to define a polymorphic formula, with respect to a given context, as illustrated in Example 6.4. In this example, since we only *write* a typed formula, the global context alone is sufficient. The next section aims to allow us to actually *type* it.

Type

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{(type variable)} \\ & | & T(\tau_1, \ldots, \tau_m) & \text{(type constructor application)} \end{array}$$

Type Scheme

$$\begin{array}{rcll} \sigma & ::= & \Pi\alpha_1, \ldots, \alpha_m.\ \tau_1 \times \cdots \times \tau_n \to \tau & \text{(function type signature)} \\ & | & \Pi\alpha_1, \ldots, \alpha_m.\ \tau_1 \times \cdots \times \tau_n \to o & \text{(predicate type signature)} \end{array}$$

Term

$$\begin{array}{rcll} t & ::= & x & \text{(variable)} \\ & | & f(\tau_1, \ldots, \tau_m; t_1, \ldots, t_n) & \text{(function application)} \end{array}$$

Formula

$$\begin{array}{rcll} A & ::= & \top & \text{(true)} \\ & | & \bot & \text{(false)} \\ & | & A_1 \wedge A_2 & \text{(conjunction)} \\ & | & A_1 \vee A_2 & \text{(disjunction)} \\ & | & A_1 \Rightarrow A_2 & \text{(implication)} \\ & | & A_1 \Leftrightarrow A_2 & \text{(equivalence)} \\ & | & P(\tau_1, \ldots, \tau_m; t_1, \ldots, t_n) & \text{(predicate application)} \\ & | & \exists x : \tau.\ A & \text{(existential quantifier)} \\ & | & \forall x : \tau.\ A & \text{(universal quantifier)} \end{array}$$

Type-Quantfied Formula

$$\begin{array}{rcll} A_T & ::= & A & \text{(formula)} \\ & | & \forall\alpha.\ A_T & \text{(type quantification)} \end{array}$$

Figure 6.3: Syntactic categories of polymorphic first-order logic.

Local Context

Γ_L	::=	$\emptyset$	(empty context)
	|	$\Gamma_L,\ \alpha : \mathsf{Type}$	(type variable declaration)
	|	$\Gamma_L,\ x : \tau$	(term variable declaration)

Global Context

Γ_G	::=	$\emptyset$	(empty context)
	|	$\Gamma_G,\ T :: m$	(type constructor declaration)
	|	$\Gamma_G,\ f : \sigma$	(function declaration)
	|	$\Gamma_G,\ P : \sigma$	(predicate declaration)

Figure 6.4: Contexts for polymorphic first-order logic.

Example 6.4: Polymorphism

Let us consider the following global context:

- A type constructor `list`: Type → Type.
- A polymorphic function `cons`: $\Pi\alpha.\ (\alpha \times \texttt{list}(\alpha)) \to \texttt{list}(\alpha)$.
- A polymorphic function `head`: $\Pi\alpha.\ \texttt{list}(\alpha) \to \alpha$.
- A polymorphic predicate `Equal`: $\Pi\alpha.\ (\alpha \times \alpha) \to o$.

The previous context allows us to build the following formula:

$$\forall\alpha\forall(x:\alpha)(\ell,\ell':\texttt{list}(\alpha)).$$
$$\texttt{Equals}(\texttt{list}(\alpha);\ell',\texttt{cons}(\alpha;\ell,x)) \Rightarrow \texttt{Equals}(\alpha;\texttt{head}(\alpha;\ell'),x)$$

6.2.2 Typing Process and Inference Rules

Once a type has been assigned to each term and predicate, we want to expand the scope to cover the entire formula. Unlike the preceding elements, formulas do not come with an explicitly declared type signature. Instead, the type must be inferred based on the inner predicates within the formulas. This mechanism is called *typing*.

The typing mechanism uses a set of inferences rules and a context. Informally, this mechanism decomposes the formula and collects values for the type of terms and type variables until reaching a predicate. At this point, it checks that the inferred type matches with the type schemes given by the context.

A context that is consistent is called *well-formed*, denoted $\texttt{wf}(\Gamma)$, and the corresponding rules are given in Figure 6.5. Analogously to this notion, a term or formula that successfully passes the typing phase (w.r.t. a well-formed context Γ) is called *well-typed*. The rules used to type a formula are presented in Figure 6.6. In the case of TFF0 and TFF1, the typing of a formula for a given context is decidable.

Example 6.5 illustrates the typing of the formulas $\exists(x, y : \texttt{int}).\ P(x, y)$ for a given context Γ. In this example, the initial global context Γ_G is composed of int of type Type and a predicate P with a signature int × int → o. The local context Γ_L, initially empty, becomes inhabited by x and y of type int at the end of the typing process.

$$\frac{}{\mathtt{wf}(\emptyset;\emptyset)}\ \mathrm{WF}_1 \qquad \frac{x \notin \Gamma_L \qquad \Gamma_G;\Gamma_L \vdash \tau : \mathsf{Type}}{\mathtt{wf}(\Gamma_G;\Gamma_L, x:\tau)}\ \mathrm{WF}_2$$

$$\frac{\alpha \notin \Gamma_L \qquad \mathtt{wf}(\Gamma_G;\Gamma_L)}{\mathtt{wf}(\Gamma_G;\Gamma_L, \alpha : \mathsf{Type})}\ \mathrm{WF}_3 \qquad \frac{T \notin \Gamma_G \qquad \mathtt{wf}(\Gamma_G;\emptyset)}{\mathtt{wf}(\Gamma_G;T :: m;\emptyset)}\ \mathrm{WF}_4$$

$$\frac{f \notin \Gamma_G \qquad \Gamma_G;\alpha_1 : \mathsf{Type},\ldots,\alpha_m : \mathsf{Type} \vdash \tau_{i_{1\le i\le n}} : \mathsf{Type} \qquad \Gamma_G;\alpha_1 : \mathsf{Type},\ldots,\alpha_m : \mathsf{Type} \vdash \tau : \mathsf{Type}}{\mathtt{wf}(\Gamma_G, f : \Pi\alpha_1 \ldots \alpha_m.\ \tau_1 \times \cdots \times \tau_n \to \tau;\emptyset)}\ \mathrm{WF}_5$$

$$\frac{P \notin \Gamma_G \qquad \Gamma_G;\alpha_1 : \mathsf{Type},\ldots,\alpha_m : \mathsf{Type} \vdash \tau_{i_{1\le i\le n}} : \mathsf{Type}}{\mathtt{wf}(\Gamma_G, P : \Pi\alpha_1 \ldots \alpha_m.\ \tau_1 \times \cdots \times \tau_n \to o;\emptyset)}\ \mathrm{WF}_6$$

Figure 6.5: Contexts for polymorphic first-order logic.

$$\frac{\alpha : \mathsf{Type} \in \Gamma \qquad \mathtt{wf}(\Gamma)}{\Gamma \vdash \alpha : \mathsf{Type}}\ \mathrm{Var_T} \qquad \frac{x : \tau \in \Gamma \qquad \mathtt{wf}(\Gamma)}{\Gamma \vdash x : \tau}\ \mathrm{Var}$$

$$\frac{T :: m \in \Gamma \qquad \Gamma \vdash \tau_{i_{(1 \leq i \leq n)}} : \mathsf{Type}}{\Gamma \vdash T(\tau_1, \ldots, \tau_n) : \mathsf{Type}}\ \mathrm{Constr_T}$$

$$\frac{\mathtt{wf}(\Gamma)}{\Gamma \vdash \top : o}\ \top \qquad \frac{\mathtt{wf}(\Gamma)}{\Gamma \vdash \bot : o}\ \bot \qquad \frac{\Gamma \vdash A : o}{\Gamma \vdash \neg A : o}\ \neg$$

$$\frac{\Gamma \vdash A_1 : o \qquad \Gamma \vdash A_2 : o}{\Gamma \vdash A_1 \wedge A_2 : o}\ \wedge \qquad \frac{\Gamma \vdash A_1 : o \qquad \Gamma \vdash A_2 : o}{\Gamma \vdash A_1 \vee A_2 : o}\ \vee$$

$$\frac{\Gamma \vdash A_1 : o \qquad \Gamma \vdash A_2 : o}{\Gamma \vdash A_1 \Rightarrow A_2 : o}\ \Rightarrow \qquad \frac{\Gamma \vdash A_1 : o \qquad \Gamma \vdash A_2 : o}{\Gamma \vdash A_1 \Leftrightarrow A_2 : o}\ \Leftrightarrow$$

$$\frac{\Gamma, x : \tau \vdash A : o}{\Gamma \vdash \exists x : \tau . A : o}\ \exists \qquad \frac{\Gamma, x : \tau \vdash A : o}{\Gamma \vdash \forall x : \tau . A : o}\ \forall \qquad \frac{\Gamma, \alpha : \mathsf{Type} \vdash A_T : o}{\Gamma \vdash \forall \alpha . A_T : o}\ \forall_T$$

$$\frac{\begin{array}{c} f : \Pi \alpha_1 \ldots \alpha_n . \\ \tau_1 \times \cdots \times \tau_n \to \tau \in \Gamma \end{array} \qquad \Gamma \vdash \tau'_{i_{1 \leq i \leq m}} : \mathsf{Type} \qquad \Gamma \vdash t_i : \rho(\tau_i)_{(1 \leq i \leq n)}}{\Gamma \vdash f(\tau'_1, \ldots, \tau'_m . t_1, \ldots, t_n) : \rho(\tau)}\ \mathrm{Fun}$$

$$\frac{\begin{array}{c} P : \Pi \alpha_1 \ldots \alpha_n . \\ \tau_1 \times \cdots \times \tau_n \to o \in \Gamma \end{array} \qquad \Gamma \vdash \tau'_{i_{1 \leq i \leq m}} : \mathsf{Type} \qquad \Gamma \vdash t_i : \rho(\tau_i)_{(1 \leq i \leq n)}}{\Gamma \vdash P(\tau'_1, \ldots, \tau'_m . t_1, \ldots, t_n) : o}\ \mathrm{Pred}$$

$$\text{With } \rho = [\alpha_1 \mapsto \tau'_1, \ldots, \alpha_m \mapsto \tau'_m]$$

Figure 6.6: Typing rules for polymorphic first-order logic.

Example 6.5: Formula Typing Process

$$\Gamma = \{\texttt{int} : \texttt{Type}, P : \texttt{int} \times \texttt{int} \to o\}$$

π:

$$\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash x : \texttt{int}} \quad \dfrac{}{\Gamma' \vdash y : \texttt{int}}}{\Gamma' = \Gamma, (x, y : \texttt{int}) \vdash P(x, y) : o} \quad \dfrac{}{\Gamma' \vdash \texttt{int} : \texttt{Type}}}{}$$

$$\dfrac{\dfrac{}{\Gamma \vdash \texttt{int} : \texttt{Type}} \quad \dfrac{\dfrac{}{\Gamma, x : \texttt{int} \vdash \texttt{int} : \texttt{Type}} \quad \dfrac{\pi}{\Gamma, (x, y : \texttt{int}) \vdash P(x, y) : o}}{\Gamma, x : \texttt{int} \vdash \exists (y : \texttt{int}).\, P(x, y) : o}}{\Gamma \vdash \exists (x, y : \texttt{int}).\, P(x, y) : o}$$

The tableau calculus retains the well-typedness of formulas [173], under the condition to be cautious with substitutions. Consequently, once all the formulas have been initially proven to be well-typed, there is no need for further verification to be carried out.

6.2.3 Integration into an Automated Theorem Prover

Now that polymorphic types have been implemented and parsed by Goéland, our focus shifts to examining the implications of these changes for proof search. While adding types to terms is relatively straightforward, the rules governing quantifiers and the handling of type variables necessitate additional considerations.

First of all, we deal with typed variables in the same way as usual variables, i.e., we create *free type variables*, of type Type, waiting for an instantiation. We also need to be careful during type unification, to ensure the consistency of the proof.

In addition, we address the concept of *Skolemized type variables*. However, due to TFF1 standards mandating the placement of type quantifiers ahead of any other quantifiers, no free variables can precede type variables. Consequently, the type Skolem generated can only be a constant. An example of a typed proof search can be found in the illustration provided in Example 6.6.

Example 6.6: Typed Tableau Proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall(\alpha : \texttt{Type})\, \forall(x, y : \alpha).\, P(\alpha; x, y), \neg P(\tau; a, b)}{\forall(x, y : A).\, P(A; x, y)}\, \forall_T}{\forall(y : A).\, P(A; X, y)}\, \forall}{P(A; X, Y)}\, \forall}{\rho = \{A \mapsto \tau\}, \sigma = \{X \mapsto a, Y \mapsto b\}}\, \odot$$

In Goéland, the typing process operates in three distinct phases. First, during the initial typing pass, the system collects type constructors and type schemes. The `wf`-rule is then triggered on Γ_G for each addition, to ensure the consistency of the

formulas. This phase is performed iteratively during the parsing of the problem itself. Once built, the global context cannot be altered.

In the second phase, the variables are gathered in pairs (term, type) to be endowed by a type. The formulas are deconstructed thanks to the typing rule until reaching an axiomatic rule, possibly adding elements to the local context. If so, a `wf`-rule is triggered on Γ_L only with this new pair. Then, each branch passes its new information to rebuild the formula, along with the inferred type. During this phase, as there are no dependencies between branches, we leverage the concurrent nature of the prover to type each branch simultaneously. If the type-checking process succeeds, the proof search can start. Otherwise, an error is returned.

In the final phase, the proof search proceeds as usual, with an additional unification mechanism activated to ensure type consistency. Ultimately, in the case of pure first-order logic, terms and predicates are typed by default, and the type-checking pass is omitted.

6.3 Conclusion

Through 30 000 lines of code, Goéland has implemented a concurrent proof-search procedure along with multiple features that enhance its capability to tackle various problems. This chapter has described its architecture, aiming to provide a comprehensive grasp of intricate aspects and their relationship with parallelism. The various extensions address a wide range of features, from debug mode to improve competitive behavior, in order to improve the usability of Goéland.

The introduction of polymorphism into the prover has expanded its potential to handle a wider range of problems. It relies on an embodiment of type directly in the terms, and a two-phase preprocessing which parses the types and checks for the consistency of the formulas in order to (un)validate the initial typing. Once checked, and given that the tableau calculus preserves the well-typedness of formulas, the proof-search procedure can be applied as usual. It results that despite its omnipresence, the addition of types into the prover is relatively smooth. As typed problems are expected to become more prevalent in the future, this extension ensures that Goéland remains relevant and applicable in diverse contexts.

Ongoing efforts are aimed at enhancing existing functionalities, such as equality reasoning and memory management, and conducting experiments on typed problems. Additionally, a memory-shared version of Goéland is in development, along with a plug-in for arithmetic reasoning.

Chapter 7

Toward Certification: an Output for Checkable Proofs

Contents

The realm of theorem proving can be categorized into two major families: the *Automated* (ATP) and the *Interactive* (ITP) approaches. Whereas the systems in the former reason autonomously and provide an outcome regarding the truth value of a given formula, the ones in the latter act as assistants for humans, building a proof hand-in-hand. ITP systems also rely on a certified kernel to ensure the correctness of any asserted proof and often work with proofs in a sequent style.

One of the significant strengths of tableaux is their ability to produce a proof, as original tableau calculus is equivalent to sequent calculus. Then, it should be reasonable for a tableau-based theorem prover to output a proof that is checkable by a proof assistant. A variation of the sequent calculus has also been proposed, that mirrors the tableau rules: GS3 [230]. Thus, we want to translate the tableau obtained by the proof-search procedure into GS3.

However, due to the use of free variables, a free-variable tableau proof search can produce a proof slightly different from the one obtained by the usual tableau calculus, making it not trivially translatable into sequent. This difference increases with the use of some optimized variants of the proof search, i.e., the Skolemization. Indeed, most proof assistants do not accept advanced Skolemization strategies such as those implemented in automated theorem provers. It results in a need for translating free-variable tableau proof into sequent.

Translating tableau proofs into GS3 sequents appears as early as 1987 for the connection tableaux [43]. These tableaux are a clausal and restricted version of the original method, where formulas are preprocessed to apply a heuristic during the branches' exploration. Years later, [7, 9] have shown that deskolemizing functions

in inner Skolemization results in a substantial increase in the proof size. [121] also proposes a framework of proof deskolemization, but is aimed towards the resolution method and as such does not preserve the syntactic integrity of the formula. Proof deskolemizing for outer Skolemization in other sequent-based systems has also been explored in [84] and [179]. Deskolemization of δ^ε-rules have been implemented in [65], but it is done during the proof-search procedure, thus enabling an immediate one-to-one mapping over proof assistants languages and excluding the need for a proof-to-proof translation. Lastly, [66] proposes a theoretic yet generic method to translate standard tableaux into GS3 sequents. To the best of our knowledge, no work has been conducted on deskolemizing proofs using pre-inner Skolemization rules.

This section outlines an improvement of the algorithm proposed in [66] capable of translating tableau proofs into sequent proofs for various Skolemization strategies as well as its implementation into Goéland. These sequent proofs serve as a base to be translated into serval formats understandable by ITP. Such a translation toward Coq [20] and Lambdapi [4] is also offered.

7.1 From Tableau Proofs to Sequent Proofs: GS3

In the tableau method, a proof is a ground tableau, i.e., a tableau without free variables. Thus, since Goéland builds its proofs with free variables, we want to find a way to translate these proofs into a general basis, itself reusable by proof-checker: GS3. The Gentzen-Schütte calculus, presented in Figure 7.1 and conveniently arranged in a way that mirrors the tableau rules of Section 1.2.1, is a variant of the original Gentzen's sequent calculus.

GS3 differs from the method of analytic tableaux as it is read from *bottom to top* as the rules are *abductive* and make a copy of the formulas of a node instead of extending it, such as the original sequent calculus. Furthermore, a *sequent* is used to label the nodes, i.e., a two-sided system where hypotheses are at the left of the symbol $\vdash$ and conclusions at the right. Nevertheless, GS3 is also a refutational calculus and thus only the left side, the hypotheses side, is used. It then suffices to find a contradiction between hypotheses. As such, the tableau method sticks closely to this calculus, where all the processed formulas are also kept in the branch.

One noteworthy difference to highlight between both methods lies in the first-order rules, i.e., the rules involving quantifiers. GS3 functions as a non-automated focused proof system, with no involvement of free variables during the proof search. Consequently, it follows the rules of ground tableaux, instantiating variables to *known* terms when dealing with γ-equivalent rules and Skolemizing with *fresh* constants.

This system is closely related to the usual systems implemented in interactive theorem provers [40, 182, 193] and can thus easily be embedded inside such tools. The main properties that those systems share is the way of dealing with quantifiers and, as such, a translation of Skolemized formulas has to be devised in order to formally certify tableau proofs.

However, as discussed in Section 1.2.2, ground tableaux provide an elegant way to present a proof, whereas free-variables tableaux are built to efficiently search for one. As a result, the proofs generated by these two tableau systems may differ, particularly

$$\frac{}{\Delta, \bot \vdash}\,\mathrm{ax} \qquad \frac{}{\Delta, \neg\top \vdash}\,\mathrm{ax} \qquad \frac{}{\Delta, F, \neg F \vdash}\,\mathrm{ax} \qquad \frac{\Delta \vdash}{\Delta, F \vdash}\,\mathrm{w}$$

(a) Structural and axiomatic rules.

$$\frac{\Delta, \neg\neg F, F \vdash}{\Delta, \neg\neg F \vdash}\,\neg_\neg \qquad \frac{\Delta, F \wedge G, F, G \vdash}{\Delta, F \wedge G \vdash}\,\wedge$$

$$\frac{\Delta, \neg(F \vee G), \neg F, \neg G \vdash}{\Delta, \neg(F \vee G) \vdash}\,\neg_\vee \qquad \frac{\Delta, \neg(F \Rightarrow G), F, \neg G \vdash}{\Delta, \neg(F \Rightarrow G) \vdash}\,\neg_\Rightarrow$$

(b) Non-branching propositional rules.

$$\frac{\Delta, F \vee G, F \vdash \qquad \Delta, F \vee G, G \vdash}{\Delta, F \vee G \vdash}\,\vee \qquad \frac{\Delta, F \Rightarrow G, \neg F \vdash \qquad \Delta, F \Rightarrow G, G \vdash}{\Delta, F \Rightarrow G \vdash}\,\Rightarrow$$

$$\frac{\Delta, \neg(F \wedge G), \neg F \vdash \qquad \Delta, \neg(F \wedge G), \neg G \vdash}{\Delta, \neg(F \wedge G) \vdash}\,\neg_\wedge$$

$$\frac{\Delta, F \Leftrightarrow G, \neg F, \neg G \vdash \qquad \Delta, F \Leftrightarrow G, F, G \vdash}{\Delta, F \Leftrightarrow G \vdash}\,\Leftrightarrow$$

$$\frac{\Delta, \neg(F \Leftrightarrow G), F, \neg G \vdash \qquad \Delta, \neg(F \Leftrightarrow G), \neg F, G \vdash}{\Delta, \neg(F \Leftrightarrow G) \vdash}\,\neg_\Leftrightarrow$$

(c) Branching propositional rules.

$$\frac{\Delta, \exists x.\, F, F[x \mapsto c] \vdash}{\Delta, \exists x.\, F \vdash}\,\exists \qquad \frac{\Delta, \neg\forall x.\, F, \neg F[x \mapsto c] \vdash}{\Delta, \neg\forall x.\, F \vdash}\,\neg_\forall$$

(d) Skolemization rules, where c is a fresh constant.

$$\frac{\Delta, \forall x.\, F, F[x \mapsto t] \vdash}{\Delta, \forall x.\, F \vdash}\,\forall \qquad \frac{\Delta, \neg\exists x.\, F, \neg F[x \mapsto t] \vdash}{\Delta, \neg\exists x.\, F \vdash}\,\neg_\exists$$

(e) Instantiation rules, where t is a ground term.

Figure 7.1: Rules of the GS3 calculus.

in terms of instantiation choices. Hence, the main challenge of this section is to translate a free-variable tableaux proof into a GS3 proof.

7.2 The Challenges of a Proof Translation

The standard free-variable tableau calculus uses *outer Skolemization* for δ-rules, as illustrated in Figure 7.2a. This strategy makes the final tableau totally equivalent to an original tableau [124] and therefore to a GS3 proof, the processing needed for a translation being thus minimal (reduced to only a one-to-one mapping between the rules of the two systems). Such a translation is available in Figure 7.3, in which $f(X)$ has been replaced by a constant c.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y))}{\neg(D(X) \Rightarrow \forall y\, D(y))}\,\gamma_{\neg\exists}}{D(X), \neg(\forall y\, D(y))}\,\alpha_{\neg\Rightarrow}}{\neg D(f(X))}\,\delta_{\neg\forall}}{\neg(D(f(X)) \Rightarrow \forall y\, D(y))}\,\gamma_{\neg\exists}}{D(f(X)), \neg\forall y\, D(y)}\,\alpha_{\neg\Rightarrow}}{\odot}\,\odot$$

(a) Outer Skolemization tableau.

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y))}{\neg(D(X) \Rightarrow \forall y\, D(y))}\,\gamma_{\neg\exists}}{D(X), \neg(\forall y\, D(y))}\,\alpha_{\neg\Rightarrow}}{\neg D(c)}\,\delta^{+}_{\neg\forall}}{\sigma = \{X \mapsto c\}}\,\odot_{\sigma}$$

(b) Inner Skolemization tableau.

Figure 7.2: Proof of the drinker paradox in outer and inner Skolemization.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \ldots, \neg D(c), \neg(D(c) \Rightarrow \forall y\, D(y)), D(c), \neg(\forall y\, D(y)) \vdash}\,\text{ax}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \ldots, \neg D(c), \neg(D(c) \Rightarrow \forall y\, D(y)) \vdash}\,\neg_{\Rightarrow}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \ldots, D(X), \neg(\forall y\, D(y)), \neg D(c) \vdash}\,\neg_{\exists}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(D(X) \Rightarrow \forall y\, D(y)), D(X), \neg(\forall y\, D(y)) \vdash}\,\neg_{\forall}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(D(X) \Rightarrow \forall y\, D(y)) \vdash}\,\neg_{\Rightarrow}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)) \vdash}\,\neg_{\exists}$$

Figure 7.3: Translation into GS3 of the drinker paradox in outer Skolemization.

However, to optimize tableau proofs, various Skolemization strategies have been extensively studied. It results in several Skolemization rules, such as δ^{+}-, $\delta^{+^{+}}$-, δ^{*}-, $\delta^{*^{*}}$- and δ^{ε}-rules [8, 36, 81, 137, 144], where the most optimized strategies δ^{ε} and $\delta^{*^{*}}$ have been shown to yield proofs shorter (for a number n of branches) by a factor of $2^{2^{2^n}}$ compared to standard Skolemization. While this is advantageous for proof-search procedures, it presents challenges when translating tableau proofs for certification.

For instance, a proof in the δ^{+}-rules (*inner* Skolemization rules), the weakest amelioration over standard tableaux, is developed in Figure 7.2b. Even though this proof does not generate any branch, it is already shorter than its outer-Skolemization counterpart. However, this makes the proof not readily translatable into a GS3 sequent. Figure 7.4 is an attempt at a naive translation that fails when the rule denoted ($\star$) is applied.

The problem arises because, in Skolemization, the resulting constant must be *fresh*. In this case, c is introduced by the $\neg\exists$ rule, and if the constant is made fresh (e.g., if the rule $\neg\forall$ yields c'), then the axiomatic rule cannot be applied because the closure in the tableau proof occurs between $D(c)$ and $\neg D(c)$, while the GS3 sequent's node will be labeled with $D(c)$ and $\neg D(c')$. To certify tableau proofs with smart Skolemization strategies, a transformation into GS3 sequents must be performed.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(D(c) \Rightarrow \forall y\, D(y)), D(c), \neg(\forall y\, D(y)), \neg D(c) \vdash}\,\text{ax}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(D(c) \Rightarrow \forall y\, D(y)), D(c), \neg(\forall y\, D(y)) \vdash}\,\neg_\forall\ (\star)}{\neg(\exists x\, (D(x) \Rightarrow \forall y\, D(y))), \neg(D(c) \Rightarrow \forall y\, D(y)) \vdash}\,\neg_\Rightarrow}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)) \vdash}\,\neg_\exists$$

Figure 7.4: Incorrect proof yielded by a naive translation in GS3 of the tableau proof of the drinker paradox in inner Skolemization.

7.3 A Deskolemization Strategy

As it has been mentioned in the previous section and highlighted in Figure 7.2b, a tableau proof is sound even if a free variable is instantiated by a term that does not (yet) exist in the said proof. However, the naive translation of such a proof into a GS3 sequent is impossible, as the freshness condition of the Skolemization rules $(\exists, \neg\forall)$ does not hold. The algorithm developed in this section overcomes this problem by offering an on-the-fly translation that refines the algorithm of [66]. This improved algorithm should generalize well to other Skolemization strategies without relying on syntactic preprocessing of formulas, as seen in [43].

The idea behind the algorithm of [66] is to, given a closed tableau T, build a sequent by following the rules executed from the root of T until reaching a δ^+-rule applied over a formula D. The sequent is then adjusted to retain only the formula D and the initial formula to prove while discarding everything else. In practice, these formulas are weakened and grafted back after the application of the δ^+-rule. This processing ensures that every δ^+-rule needed in a branch is applied first, before any other rule. This approach is realized through a "grow, weaken, and graft" strategy, as illustrated in Figure 7.5.

However, the previously described version of the algorithm is pathologically inefficient due to the weakening-and-grafting strategy incurred when encountering a δ^+-rule. As proofs in inner-Skolemization gain an exponential number of branches over the GS3 proof, this strategy *always* creates a sequent that is exponentially bigger than the original proof. It is however possible to craft an on-the-whole better translation that is in the worst case the same as previously presented but much better in average. For this, we need to introduce the notions of *dependency* and *descendants*.

Definition 7.1: Dependency

Let D be a formula on which a δ^+-rule can be applied, and δ_D the Skolem symbol yielded by the application of the rule on D. Let Γ_γ be the set of formulas on which a γ-rule can be applied. A formula $F \in \Gamma_\gamma$ *depends* on D if and only if $F \hookrightarrow F'$ and there exists ω such that $F'_{|\omega} = \delta_D$ (i.e., the subterm at the index ω of F' is δ_D). The set of formulas which depend on D is denoted $\Delta(D)$ and

defined as follows:

$$\Delta(D) = \left\{ F \in \Gamma_\gamma \mid F \hookrightarrow F' \wedge \exists \omega . F'_{|\omega} = \delta_D \right\}$$

This set allows the algorithm to know exactly which formulas introduce a forbidden Skolem symbol δ_D and thus to have a starting point to subsequently select the formulas that need to be weakened, where the previous algorithm would always weaken everything. To keep the number of formulas weakened to a minimum, this set must then be extended by adding solely the formulas *descended* from the formulas in $\Delta(D)$ that have an occurrence of δ_D.

Definition 7.2: Descendance

Let F be a formula of a leaf L of a tableau. Another formula $G \in L$ is said to be *descending* from F if and only if $F \hookrightarrow F_1 \hookrightarrow \cdots \hookrightarrow F_n$ and there exists $k \leq n$ such that $G = F_k$. If there exists D such that $F \in \Delta(D)$, then the set of formulas descending from F which are also dependant on D is denoted $\Lambda(F)$ and defined as follows:

$$\Lambda(F) = \left\{ G \in L \mid F \hookrightarrow^* G \wedge \exists \omega . G_{|\omega} = \delta_D \right\}$$

Let us note that if D is descending from a formula F in $\Delta(D)$, then the algorithm fails (or does not terminate). Fortunately, it can not happen as stated in the following lemma.

Lemma 7.3: No Self Dependency

Let D be $\exists x . D'$ or $\neg \forall x . D'$, i.e., a δ^+-rule can be applied to D. Then forall $F \in \Delta(D)$, $D'[x \mapsto \delta_D]$ is not in $\Lambda(F)$.

Proof. Let us suppose that there exists $F \in \Delta(D)$ such that $D'[x \mapsto \delta_D] \in \Lambda(F)$. By determinism of the tableau rules application, if $D'[x \mapsto \delta_D] \in \Lambda(F)$, then $D \in \Lambda(F)$ and so there exists ω such that $D_{|\omega} = \delta_D$. Recall that in inner Skolemization, the free variables occurring in a formula are taken as arguments of the Skolem symbol yielded by the rule. By definition of the dependency, it means that δ_D should be a parameter of the symbol returned by the δ^+-rule over D and thus there exists ω such that $\delta_{D|\omega} = \delta_D$, which can not happen by definition of a term's construction in first-order logic. □

The main ideas behind the new algorithm are the same as the ones behind the previous algorithm, i.e., follow the tableau proof by seamlessly applying the GS3 rule corresponding to the tableau rule while the latter is not a δ^+-rule. Once it is a δ^+-rule, weaken the relevant formulas and make the tree grow back to its pre-weakened state. This last step is now called *growing back* instead of *grafting*, as the pre-weakened tree can not simply be *grafted* back now that only some picked-out formulas are weakened. As such, the algorithm operates as follows:

$$\dfrac{\dfrac{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(D(c) \Rightarrow \forall y\, D(y)), D(c), \neg(\forall y\, D(y)) \vdash}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(D(c) \Rightarrow \forall y\, D(y)) \vdash}\,\neg_{\Rightarrow}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)) \vdash}\,\neg_{\exists}$$

(a) First steps of the proof.

$$\dfrac{\dfrac{\dfrac{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(\forall y\, D(y)) \vdash}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), D(c), \neg(\forall y\, D(y)) \vdash}\,\mathrm{w}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(D(c) \Rightarrow \forall y\, D(y)), D(c), \neg(\forall y\, D(y)) \vdash}\,\mathrm{w}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(D(c) \Rightarrow \forall y\, D(y)) \vdash}\,\neg_{\Rightarrow}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)) \vdash}\,\neg_{\exists}$$

(b) Cleaning the relevant formulas descending from $\Delta(D)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(D(c) \Rightarrow \forall y\, D(y)), D(c), \neg(\forall y\, D(y)), \neg D(c) \vdash}\,\mathrm{ax}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(D(c) \Rightarrow \forall y\, D(y)), \neg(\forall y\, D(y)), \neg D(c) \vdash}\,\neg_{\Rightarrow}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(\forall y\, D(y)), \neg D(c) \vdash}\,\neg_{\exists}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(\forall y\, D(y)) \vdash}\,\neg_{\forall}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), D(c), \neg(\forall y\, D(y)) \vdash}\,\mathrm{w}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(D(c) \Rightarrow \forall y\, D(y)), D(c), \neg(\forall y\, D(y)) \vdash}\,\mathrm{w}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)), \neg(D(c) \Rightarrow \forall y\, D(y)) \vdash}\,\neg_{\Rightarrow}}{\neg(\exists x.\, D(x) \Rightarrow \forall y\, D(y)) \vdash}\,\neg_{\exists}$$

(c) Skolemization, applying back $\mathscr{R}$'s rules and finalisation.

Figure 7.5: Sound translation into GS3 of the drinker paradox in inner Skolemization using the algorithm.

1. While the rule applied is not a δ^+-rule, apply the GS3 rule corresponding to the tableau rule and mark the formula on which it is applied (Figure 7.5a).
2. Let D be the formula on which the δ^+-rule is applied.
3. For every formula F in $\Delta(D)$, weaken the sequent to remove all marked formulas of $\Lambda(F)$ and record the rules used to derive these formulas in their application order in a set called $\mathscr{R}$ (Figure 7.5b).
4. Apply the δ^+-rule on D (first step of Figure 7.5c).
5. Apply back the rules recorded in $\mathscr{R}$ (last two steps of the Figure 7.5c).
6. Repeat while a rule is applied in the corresponding tableau leaf.

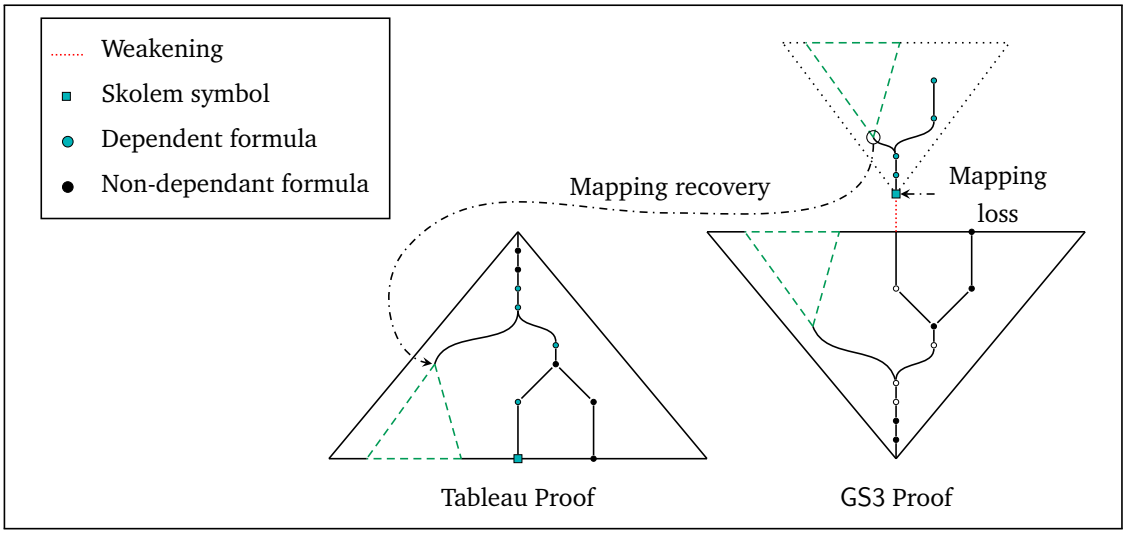


Figure 7.6: Mapping conservation after reapplication of a β-rule.

Specific attention is paid to β-rules in step 5. When a dependent formula has a β-rule applied to it, it generates two branches: one for the Skolem symbol that initiates the weakening and another. To continue the translation on the first branch, applying the remaining weakened rules is sufficient. However, the second branch is "lost" because applying the previously weakened formula does not lead to its closure. Instead, the branch must follow the path of the other branch from the original split. This behavior is ensured through a global correspondence between branches, as illustrated in Figure 7.6.

This improved algorithm constructs a sequent from a given tableau proof, with deskolemization steps applied only to the relevant formulas.

7.4 Soundness of the Translation over Inner Skolemization

The idea of the soundness proof is to build a *correspondence function*, which will be subsequently called *mapping*, between a valid GS3 sequent and the reference tableau proof. The function's domain will be total, thus associating every leaf of a valid sequent to a leaf of the reference tableau, and follows closely the definition of the algorithm. As the mapping is conserved all along the execution of the later, it ensures that when it terminates, the sequent yielded is valid and that all its leaves are closed. This section starts by formally defining the notions that will be needed to prove the soundness theorem and continues directly by developing the proof.

After finishing its proof-search procedure, Goéland yields a tableau proof of a formula F. This proof will then be used as a reference to subsequently build the GS3 sequent corresponding to the proof of F. As the goal is to make each leaf of the under-construction sequent correspond to a leaf of the reference tableau, it is necessary to define the intermediate tableaux with relevant leaves, as it is difficult to match the leaves of a partial proof and the leaves of a final proof.

Definition 7.4: Initial Part

Let T be a tableau proof. T_0 is an *initial part* of T if and only if T_0 and T share the same root, the same rule is applied to this very root and all the children of T_0 are *initial parts* of the corresponding children in T.

The notion of initial part is defined between tableaux, but it can be extended seamlessly to GS3 sequents as it is defined in the exact same way. Furthermore, both notions of *leaf* and *initial segment* are also shared definitions between tableaux and sequents, with a *leaf* intuitively being the last node of a branch identified by the set of formulas that labels it and the *initial segment* being the analogy of an initial part but for branches, i.e., an initial segment of a branch is a prefix of this very branch. The notion of *leaf* is extended to all the branches of a tableau T or a sequent π, with the set of all leaves being denoted $\mathcal{L}(T)$ or $\mathcal{L}(\pi)$. Let us now formally define the correspondence function between a tableau and a sequent, called the *mapping*.

Definition 7.5: Mapping

Let T be a tableau proof and π a GS3 sequent. A mapping $\mu : \mathcal{L}(\pi) \to \mathcal{L}(T)$ is a total function which, for every leaf $L \in \pi$ associates a leaf in T such that $L \supseteq \mu(L)$.

As previously done, the notion of *mapping* can be seamlessly extended between two GS3 sequents. For ease of understanding, a mapping between a GS3 sequent and a tableau will often be denoted μ while a mapping between two GS3 sequents will be denoted λ. The inclusion condition between a leaf and its preimage is useful then, as otherwise, for δ^+ rules, there is a direct identity between a leaf of the tableau and a leaf of the sequent.

Theorem 7.6: Soundness

Let T be a tableau proof of a formula F. Then the algorithm of Section 7.3 yields a sound GS3 proof.

Proof. By definition, the algorithm properly orders the application of δ-rule for them to yield *fresh* constants when they are applied in the sequent. Indeed, suppose that the Skolem symbol is not fresh, thus there should exist a formula G such that δ_D appears in G when the δ-rule is applied to D. Furthermore, all descendants of a formula depending on δ_D are weakened before applying the δ-rule, thus G is not a descendant of any such formula. So either δ_D is a *ground* term, either it has been generated on the application of a δ-rule on an antecedent of G. Both cases are impossible, as the symbol introduced comes from the tableau proof T which is sound by assumption. Therefore, every constant generated by a Skolemization rule in the final sequent is fresh. Let π be the GS3 proof given by the algorithm. Let $\mu : \mathcal{L}(\pi) \to \mathcal{L}(T)$ and π' respectively be the mapping and GS3 sequent given by applying the Lemma 7.7 on T (which is actually an initial part of itself). Then, for

every leaf $L \in \pi'$, $L \supseteq \mu(L)$. As $\mu(L)$ contains a contradiction, L also does and thus all the leaves of π' are closed. Furthermore, as π' is an initial part of π with all its leaves closed, by definition π' is π. Therefore, π's well and truly a sound proof of F. □

The idea behind Lemma 7.7 is to step-by-step build a mapping between the sequent generated by the algorithm and a well-chosen tableau proof (which is, in fact, an initial part of the reference tableau). In most cases, it is simple to make the mapping grow along the sequent, as almost every rule (except for a δ^+-rule) only extends it and follows the tableau proof quite literally, thus yielding a clear mapping between both proofs. However, it is not so clear for δ^+-rule and Lemma 7.9 goes over the technical details needed to retain the mapping after weakening the formulas and growing back the sequent up to its previous state.

Lemma 7.7: δ^+-Proof Mapping

> If T is a tableau proof of a formula F and π the GS3 proof generated by the algorithm, then forall initial part T_0 of T, there exists an initial part π_0 of π such that $\mu_0 : \mathcal{L}(\pi_0) \to \mathcal{L}(T_0)$ is a mapping.

Proof. Let T_0 be an initial part of T. The initial part π_0 is selected and the mapping μ_0 is built by induction on the number of rules applied in T_0, denoted $|T_0|$.

- If $|T_0| = 0$, then T_0 is composed solely of one node: the root node of the sequent. Thus π_0 also is the root node of the tableau T and π_0 trivially maps to T_0.
- If $|T_0| > 0$, then there is at least one leaf f of T_0 which is different than the root, i.e., at least one rule r has been applied to a formula φ to yield f. As such, let T_1 be T_0 without the formulas of f generated by its last rule and let f' be such a leaf. Let π_1 and $\mu_1 : \mathcal{L}(\pi_1) \to \mathcal{L}(T_1)$ respectively be the initial part of π and the mapping yielded by the induction hypothesis. π_1 can not be closed as otherwise no rule would be applicable in any leaf of T_1, and in particular, no rule would have been applied to f'. Let thus π_0 be π_1 where the GS3 rule corresponding to the tableau rule r is applied in $\mu^{-1}(f')$ on φ (which exists in this leaf, as $f' \subseteq \mu^{-1}(f')$). μ_0 is built by extending μ_1 and depends on the applied rule r. There are several possible cases.
  - r is a closure rule and in this case, $\mu_0 = \mu_1$ as no formula is added in f' and thus $f = f'$.
  - r is an α- or γ-rule and in this case, $\varphi \hookrightarrow \psi$. Thus, as ψ is also in f, the mapping $\mu_0 : \mathcal{L}(\pi_0) \to \mathcal{L}(T_0)$ is defined as follows for every leaf b of π_0:

$$\mu_0(b) = \begin{cases} f' \cup \{\psi\} & \text{if } b \text{ is } \mu^{-1}(f') \cup \{\psi\} \\ \mu_1(b) & \text{otherwise} \end{cases}$$

  - r is a β-rule and in this case, $\varphi \hookrightarrow \psi_1, \psi_2$. Thus, f' is split into two leaves f_1 and f_2 where, without loss of generality, $\psi_1 \in f_1$ and $\psi_2 \in f_2$. The

mapping $\mu_0 : \mathcal{L}(\pi_0) \to \mathcal{L}(T_0)$ is defined as follows for every leaf b of π_0:

$$\mu_0(b) = \begin{cases} f_1 & \text{if } b \text{ is } \mu^{-1}(f') \cup \{\psi_1\} \\ f_2 & \text{if } b \text{ is } \mu^{-1}(f') \cup \{\psi_2\} \\ \mu_1(b) & \text{otherwise} \end{cases}$$

– r is a δ^+-rule and then μ_0 is the mapping yielded by applying Lemma 7.9 on π_0, T_0, μ_1 and f'.

□

The preservation of the mapping is difficult for δ^+-rules, as it is first lost when applying the weakening rules, and it is regained only when the sequent has been fully grown back. It is thus necessary to prove that every leaf of the sequent provided after applying the routine still maps properly over T_0. In effect, it is not too difficult to build it for most rules, except for β-rules where the mapping should be carefully picked up from a previously generated leaf of the sequent.

Furthermore, to prove that the sequent can properly grow back up, a mapping between sequents is necessary to make correspond the work-in-progress one to the one yielded by the induction hypothesis of Lemma 7.7, π_1. But π_1 can not be directly taken as the target of the mapping, as by definition the image of a leaf by a mapping should be included in said leaf and the algorithm removes formulas from the relevant leaf. Thus, a particular initial part of π_1 has to be picked out to serve as the target.

Definition 7.8: Subsumed Initial Parts

Let π be a sequent, B be a branch of this sequent, and E a set of formulas. The subsumed initial parts of π by E is denoted $\Pi(\pi, B, E)$ and contains all the initial parts of π such that the set labeling the leaf of the branch B is included in E.

In essence, the following lemma specifies what needs to be done when applying back rules in the algorithm. It is easy for α- and γ-rules, as the branch just needs to be extended with the formula yielded by the application of this rule. It is a bit tricky for β-rules, as only one of the leaves created by its application is a prefix of the branch being grown back. Thus the other leaf needs to be mapped to the same tableau node as the node generated by the original application of the β-rule that is not a prefix of the branch being extended.

Lemma 7.9: δ^+-Rules Mapping Conservation

Let $\mu_1 : \mathcal{L}(\pi_1) \to \mathcal{L}(T_1)$ be a mapping, L be an open leaf of T_1 such that the next rule applied to L is a δ^+-rule on a formula D, π_0 be the GS3 sequent after execution of the routine on π_1 and $\mu_1^{-1}(L)$ and T_0 the tableau T_1 after application of the δ^+-rule, which generates δ_D. Then there exists a mapping $\mu_0 : \mathcal{L}(\pi_0) \to \mathcal{L}(T_0)$ which extends μ_1.

Proof. This proof is done by building families of mappings and sequents by induction on the number of δ-terms which depend on δ_D, i.e., the number of δ-terms such that δ_D is a sub-term of those very δ-terms. With the right extension, the last item of these families can then yield the desired mapping. As the rules generating these δ-terms are applied back, it is important to note that δ_D can not depend on itself (by Lemma 7.3) and that, by transitivity, it can not depend on any of its dependencies. Let π_0^0 be π_1 where the branch B carrying $\mu^{-1}(L)$ is extended by following the algorithm before applying back the rules, i.e., where all formulas of $\Lambda(D)$ are weakened and the Skolemization rule has been applied to D, generating D'. As such, let Π_1^0 be $\Pi(\pi_1, B, \mu^{-1}(L))$, i.e., the subsumed initial parts of π_1 over B and the set $\mu^{-1}(L)$. Π_1^0 can be totally ordered by inclusion and we can thus take π_1^0 to be $\max(\Pi_1^0)$, with b being its leaf on B. π_0^0 can then be mapped to π_1^0 by the function $\lambda_1^0 : \mathcal{L}(\pi_0^0) \to \mathcal{L}(\pi_1^0)$ defined as follows for every leaf n of π_0^0:

$$\lambda_1^0(n) = \begin{cases} b & \text{if } n \text{ is } (\mu^{-1}(L) \setminus \Lambda(D)) \cup \{D'\} \\ n & \text{otherwise} \end{cases}$$

We then construct by induction over the number n of formulas that have been weakened a family of mappings $(\lambda_1^i)_{i \leq n}$, initial parts $(\pi_1^i)_{i \leq n}$ and sequents $(\pi_0^i)_{i \leq n}$ as follows.

- If no δ-term depends on δ_D, then the k^{th} member of the families are built depending on the k^{th} rule r that needs to be applied back as follows:
  - r is neither a closure rule nor a δ^+-rule as no such rules depend from δ_D.
  - If r is an α- or a γ-rule generating φ, then π_0^k is π_0^{k-1} where B has been extended with φ, yielding the leaf L', Π_1^k is $\Pi(\pi_0^k, B, L')$ and thus $\pi_1^k = \max(\Pi_1^k)$. Let b be the leaf of B in π_1^k. Then λ_1^k is defined as follows for every leaf n of π_0^k:

$$\lambda_1^k(n) = \begin{cases} b & \text{if } n \text{ is } L' \\ \lambda_1^{k-1}(n) & \text{otherwise} \end{cases}$$

  - If r is a β-rule which produces φ_1 and φ_2, then let us suppose without loss of generality that $\varphi_1 \in \mu_1^{-1}(L)$. Let us define π_2 to be π_0^{k-1} where L' the leaf of B has been extended into two leaves L_1 ($\ni \varphi_1$) and L_2 ($\ni \varphi_2$), Π_1^k is $\Pi(\pi_2, B, L_1)$ and thus $\pi_1^k = \max(\Pi_1^k)$. As such, let b be the parent of B's leaf in π_1^k. The mapping of L_1 is straightforward as it is simply $b \cup \{\varphi_1\}$. However, selecting π_0^k and L_2's mapping is not. Indeed, the node $b \cup \{\varphi_2\}$ might not be a leaf, as the sequent could have been previously developed. As such, let π_3 be the sequent starting at the node $b \cup \{\varphi_2\}$ in π_1^k. π_0^k is defined to be π_2 where the subsequent tree that is rooted at L_2 becomes π_3 (therefore, if π_3 is rooted at L_3 then $L_2 \supseteq L_3$) as illustrated in Figure 7.6. Then λ_1^k is defined as follows for every leaf n of π_0^k:

$$\lambda_1^k(n) = \begin{cases} b \cup \{\varphi_1\} & \text{if } n \text{ is } L_1 \\ n & \text{if } n \text{ is a leaf of } \pi_3 \\ \lambda_1^{k-1}(n) & \text{otherwise} \end{cases}$$

- If at least one δ-term depends on δ_D, the k^{th} mapping, initial part and sequent are the same as those defined for the previous case, except for the δ^+-rule where they are seamlessly defined as the induction hypothesis directly yields $({\pi'}^i_0)_{i\leq m}$, $({\pi'}^i_1)_{i\leq m}$ and $({\lambda'}^i_1)_{i\leq m}$ and thus giving $\pi^k_1 = {\pi'}^m_1$, $\pi^k_0 = {\pi'}^m_0$ and $\lambda^k_1 = {\lambda'}^m_1$.

Finally, π^n_0 is the sequent where all the rules have been applied back and thus is π_0 and if b is its branch B's leaf, then $b \supseteq \mu_1^{-1}(L)$ and π^n_1 is π_1. As such, as $\lambda^n_1(b) \subseteq b$ and $\lambda^n_1(b) \in \mathcal{L}(\pi_1)$ then $b \supseteq \mu_1(\lambda^n_1(b))$ and μ_1 can be extended for every leaf n of π_0 as follows to yield μ_0:

$$\mu_0(n) = \begin{cases} \mu_1(\lambda^n_1(n)) \cup \{D'\} & \text{if } n \text{ is } b \\ \mu_1(\lambda^n_1(n)) & \text{otherwise} \end{cases}$$

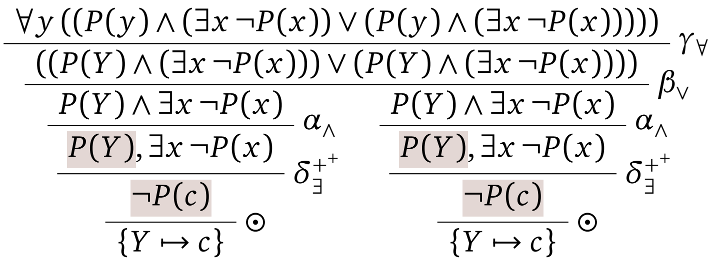
□

7.5 Extensions to δ^{+^+}

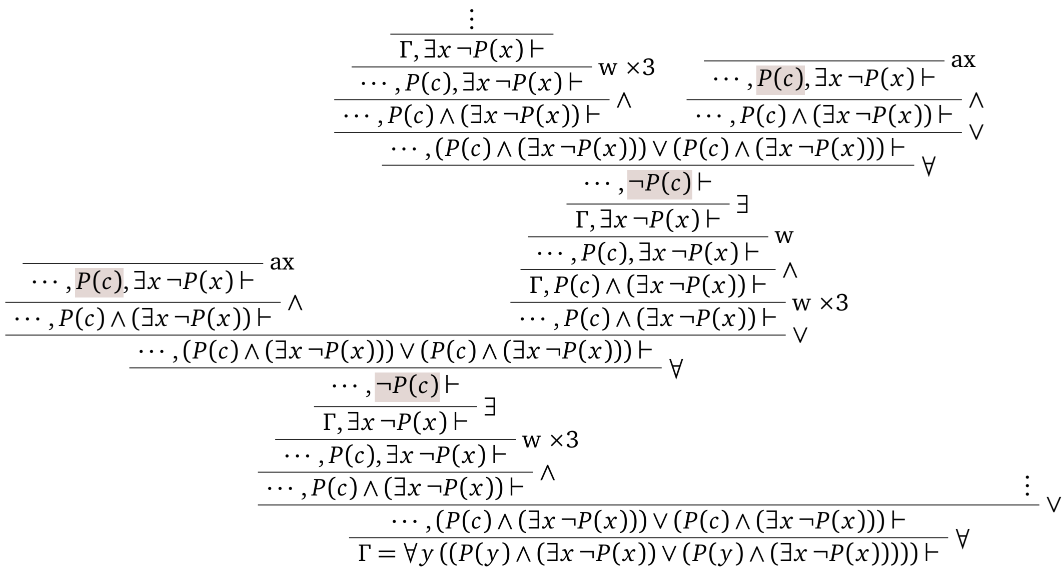
Most of the problems faced when translating δ^{+^+} tableau proofs (as introduced in Section 2.1) to GS3 sequents are properly managed by the translation algorithm introduced previously. Indeed, as those rules build over inner Skolemization, the ordering of the rules applied is naturally taken into account when deskolemizing. However, on-the-fly Skolemization generating the same symbol for α-equivalent formulas introduces a new factor in-between the original proof and the translated sequent: a γ-formula (i.e., a formula on which a γ-rule can be applied) can depend of multiple different δ-formulas. Thus, the previous algorithm needs to be adapted to handle those cases.

Figure 7.7a is the closed tableau of a formula in which both δ^{+^+}-rules give the same symbol and use it to close their respective branch, having the same final substitution. The translation of this proof by the deskolemization algorithm is illustrated in Figure 7.7b. In this example, the root formula Γ depends on the δ-formulas found in the two branches, as they generate the exact same Skolem symbol c. Applying the algorithm, when c is met in a branch, the whole formula is grafted (including the β-rule). Thus, the other branch grafts its part, including c, which triggers the algorithm again. Even if a human can immediately find a contradiction in the branch, this algorithm only *translates* a proof rather than *searches* for one. As such, when the other branch is grafted back, the Skolem symbol is detected by the algorithm and the rules of the initial tableau are applied again, leading to an infinite loop between branches.

Intuitively, when downgrading a δ^{+^+} proof to a δ^+ proof, there are two cases when applying a Skolemization rule generating a symbol δ_D: either δ_D does not yet exist and thus nothing needs to be done, either δ_D exists and as such, every formula which depends on it needs to be reintroduced. For instance, Figure 7.8 shows the proof in δ^+, which is the counterpart of the δ^{+^+} proof of Figure 7.7a. In fact, such a proof is eerily similar to the GS3 sequent given by the translation algorithm of



(a) Tableau proof yielding the same δ^{+^+}-symbol.



(b) A Diverging translation using the algorithm.

Figure 7.7: Formula that makes the translation algorithm diverge.

Figure 7.7b, except the fact that no δ-rule is applied back after reintroducing as the needed-for-closure formula ($\neg P(c_1)$) is not weakened when branching.

As such, extending the translation algorithm to δ^{+^+} proofs is fairly straightforward: it suffices to avoid weakening the Skolemized formulas and reapplying a δ^{+^+}-rule over a previously Skolemized formula. Intuitively, two cases can then happen — (i) the non-weakened Skolemized formula is useless in the branch and as such the weakening serves no purpose and (ii) the non-weakened formula should have been generated subsequently, but as it already appears in the branch with the right Skolem symbol, the mapping is preserved.

To formalize this intuition, we need to rework the β-rules case of the proof of Lemma 7.9 to properly select the sequent over which both resulting branches map to.

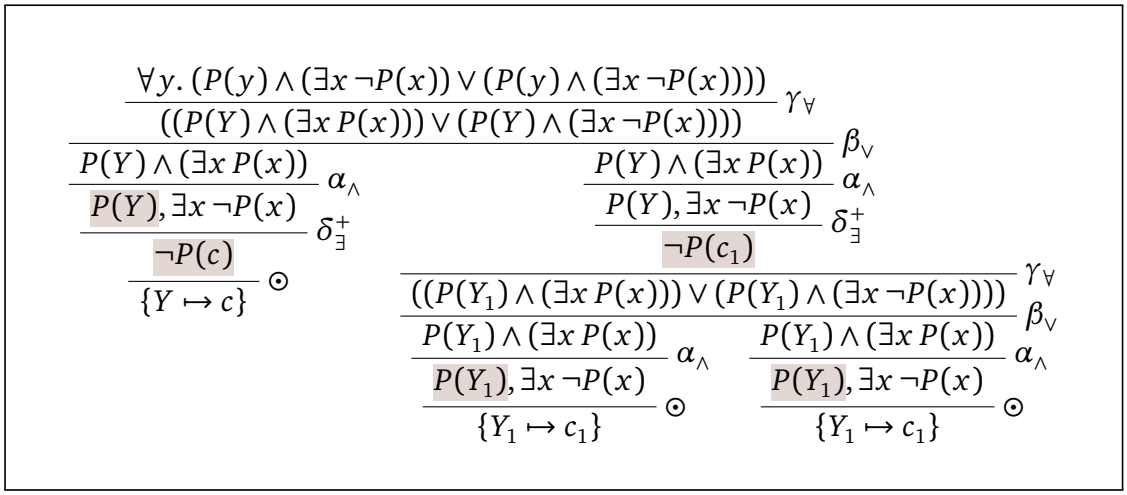


Figure 7.8: Translation of the δ^{+^+} proof to the δ^+ proof.

Lemma 7.10: δ^{+^+}-Rules Mapping Conservation

Let $\mu_1 : \mathcal{L}(\pi_1) \to \mathcal{L}(T_1)$ be a mapping, f be an open leaf of T_1 such that the next rule applied to f is a δ^{+^+}-rule on a formula D, π_0 be the GS3 sequent after execution of the routine on π_1 and $\mu_1^{-1}(f)$ and T_0 the tableau T_1 after application of the δ^{+^+}-rule, which generates the formula D' and the Skolem term δ_D. Then there exists a mapping $\mu_0 : \mathcal{L}(\pi_0) \to \mathcal{L}(T_0)$ which extends μ_1.

Proof. Recall that this proof relies on building families of (i) initial parts of π_1 denoted π_1^i, (ii) sequents that grow to become π_0 denoted π_0^i and (iii) mappings between the last elements of (i) and (ii) denoted λ_1^i. Also recall that $\mu_1^{-1}(f)$ is considered to be on the branch B in π_0^0, and, by extension, in every π_0^i. The induction cases (over the number of δ-terms which depend on δ_D) only change when applying back a β-rule. In this case, let φ_1 and φ_2 the formulas yielded by applying the β-rule in π_0^{k-1}. Without loss of generality, let us suppose that $\varphi_1 \in \mu_1^{-1}(f)$. As such, let π_2 be π_0^{k-1} where the leaf of B in π_0^{k-1} has been extended in two leaves f_1 ($\ni \varphi_1$) and f_2 ($\ni \varphi_2$). Let π_1^k be $\max(\Pi(\pi_2, B, f_1))$ and f' be $\lambda_1^{k-1}(f_1 \setminus \{\varphi_1\})$, i.e., the parent node of the leaf of B in π_1^k. Recall that $f' \cup \{\varphi_1\}$ is a leaf of π_2 as well as π_1^k so the mapping is straightforward. On the other hand, $f' \cup \{\varphi_2\}$ might not be a leaf of π_1^k, and as such let π_3 be the sequent starting at the node $\lambda_1^{k-1}(f') \cup \{\varphi_2\}$ in π_1^k. Then, let π_3' be π_3 where all the nodes have been augmented with D', i.e., if n is a node of π_3, then $n \cup \{D'\}$ is a node of π_3'. Thus it suffices to define π_0^k to be π_2 where f_2 is replaced by grafting π_3', i.e., where the subtree f_2 is replaced by the tree π_3'. As such, $\lambda_1^k : \mathcal{L}(\pi_0^k) \to \mathcal{L}(\pi_1^k)$ can be defined as follows for every leaf b of π_0^k:

$$\lambda_1^k(b) = \begin{cases} f' \cup \{\varphi_1\} & \text{if } b \text{ is } f_1 \\ b \setminus \{D'\} & \text{if } b \in \pi_3' \text{ and the corresponding} \\ & \text{node is not labelled with } D' \text{ in } \pi_3 \\ b & \text{if } b \in \pi_3 \\ \lambda_1^{k-1}(b) & \text{otherwise} \end{cases}$$

It is important to note that every leaf of π'_3 is either a leaf of π^k_1 (and as such, D' does not need to be weakened on it) or either it should be weakened of D' to be a leaf of π^k_1. In both cases, the invariant of the mapping (i.e., $b \supseteq \lambda^k_1(b)$) is preserved, and as such λ^k_1 is a mapping. □

The soundness of this algorithm can thus be directly deduced by combining this proof with the following argument of termination: in all branches, the number of formulas depending on a Skolem term not yet created decreases strictly every time the grafting routine is carried out. This argument can also be used to prove the termination of the algorithm when applied over δ^+-proofs, but has not been explicitly given as in this case, the termination is clear.

The successful extension of the algorithm to δ^{++}-rules also achieves the initial goal of building an algorithm that can serve as a solid basis for deskolemizing, even with more advanced δ-rules.

7.6 Coq and Lambdapi Output From GS3

The sequent output of Goéland is a mean towards the goal of outputting certified proofs. To achieve machine-checkable proofs, a translation process is implemented in two layers: deskolemization and translation. The deskolemization layer transforms a tableau proof into a deskolemized GS3. It is a straightforward implementation of the algorithm given in Section 7.3 and can be found in the Goéland's public repository[1].

The advantage of this deskolemization step is that it offers a sequent proof that is easily checkable by any proof-assistant, as GS3 can be embedded in most such tools. For instance, two embeddings of GS3 into Coq [20] and Lambdapi [19] have been implemented in Goéland. The translation rules for Coq can be found in Appendix A and those for Lambdapi in the branch `dev/ill/lambdapi`. The respective translations of the drinker paradox of Figure 7.5 can be found in Figure 7.9 and Figure 7.10. Most of the files consist solely of lemmas that represent the GS3 rules, which can be used almost instantly. Some similar embedding has also been implemented in Zenon [65, 83].

As Goéland makes use of deduction modulo theory to reduce the proof's size, it is normal to consider an output for proofs with rewrite rule steps. The GS3 translation captures the rewrite rules and puts them into the axioms. Their management then depends on the chosen system: the rewrite rules are kept for the Coq's translation, whereas they are omitted for the Lambdapi's one, as this system natively manages proofs in deduction modulo theory.

The translated proofs can be offered to their respective proof assistant to be formally certified. Moreover, the layer of abstraction added by the GS3 sequent allows us to easily translate Goéland's proofs into the language of any proof-assistant, validating the genericity and reusability of the method.

[1] in the folder `plugins/gs3`

```
Parameter goeland_U : Set. (* goeland's universe. *)
Parameter goeland_I : goeland_U. (* an individual in the universe. *)

Parameter d : (goeland_U -> Prop).
Parameters X : goeland_U.
Theorem goeland_proof_of_problems_p :
    ~(~((exists (X : goeland_U),
        ((d(X) -> (forall (Y : goeland_U), (d(Y)))))))).
Proof.
intro H0. apply H0. exists ((X)). apply NNPP. intros H1.
apply (goeland_notimply_s _ _ H1). intros H2 H3.
apply H3. intros skolem_Y. apply NNPP. intros H4.
apply H0. exists (skolem_Y). apply NNPP. intros H5.
apply (goeland_notimply_s _ _ H5). intros H6 H7.
auto.
Qed.
```

Figure 7.9: Coq proof of the drinker paradox, translated from the GS3 output.

7.7 Conclusion

While tableau methods yield proofs, obtaining certified proof is not always guaranteed, especially with more optimized proof-search strategies. The deskolemization mechanism, by ensuring proof equivalence, allows for efficient proof search while retaining the key advantages of tableau methods: producing a proof. In this way, the ATP becomes a proof-certificate generator [18, 178], which is later checked by an external proof checker.

By delegating the verification task to an external proof checker, we significantly enhance the level of trust we place in these proofs. It also establishes a common language for expressing proof, which allows to combine proofs generated by various theorem provers with different systems. Additionally, rather than implementing a deskolemization process in every automated theorem prover, a standardized format for outputting tableau proofs could be developed. These proofs could then be processed by a dedicated tool implementing the translation algorithm within a certified environment such as Coq or Lambdapi, enabling automatic certification of tableau proofs.

```
symbol d : τ(ι) → Prop;
symbol X : τ(ι);

symbol goeland_problems_p :
∈ ¬((∃α (λ (v0 : τ(ι)),
(d(v0) ⇒ (∀α(λ (v1 : τ(ι)), d(v1)))))))  → ∈ ⊥ ≔
λ (v2 : ∈ ¬((∃α (λ (v0 : τ(ι)),
(d(v0) ⇒ (∀α(λ (v1 : τ(ι)), d(v1))))))),
GS3nex
(ι)
(λ (v0 : τ(ι)), (d(v0) ⇒ (∀α(λ (v1 : τ(ι)), d(v1)))))
(X)
(   λ (v3 : ∈ (¬((d(X) ⇒ (∀α(λ (v1 : τ(ι)), d(v1)))))),
GS3nimp
(d(X))
((∀α(λ (v1 : τ(ι)), d(v1))))
(   λ (v4 : ∈ (d(X))),
λ (v5 : ∈ (¬((∀α(λ (v1 : τ(ι)), d(v1)))))),
GS3nall
(ι)
(λ (v1 : τ(ι)), d(v1))
(   λ (v6 : τ(ι)),
λ (v7 : ∈ (¬(d(v6)))),
GS3nex
(ι)
(λ (v0 : τ(ι)), (d(v0) ⇒ (∀α(λ (v1 : τ(ι)), d(v1)))))
(v6)
(   λ (v8 : ∈ (¬((d(v6) ⇒ (∀α(λ (v1 : τ(ι)), d(v1)))))),
GS3nimp
(d(v6))
((∀α(λ (v1 : τ(ι)), d(v1))))
(   λ (v9 : ∈ (d(v6))),
λ (v5 : ∈ (¬((∀α(λ (v1 : τ(ι)), d(v1)))))),
GS3axiom (d(v6)) (v9) (v7)
)
(v8)
)
(v2)
)
(v5)
)
(v3)
)
(v2);
```

Figure 7.10: Lambdapi proof of the drinker paradox, translated from the GS3 output.

Chapter 8
Experiments and Analysis

Contents

This section presents the performances of Goéland on the Thousand of Problems for Theorem Provers (TPTP) library [226][1] (v8.1.2). This library is the reference for testing the developed tools as it has developed a standardized way to represent logical problems and features over nine thousand (first-order logic) problems, ranging from syntactic theorems to industrial proof obligations.

To begin, we conduct a comparative analysis of all the different variants of Goéland, aiming to identify the most effective combination. Subsequently, we evaluate these selected variants against state-of-the-art theorem provers.

We also perform scalability tests to investigate how Goéland behaves under various core configurations. Lastly, we examine the impact of the transformations discussed in Section 7 by comparing the sizes of proofs before and after the treatment.

The experiments have been done on an Intel Xeon E5-2680 v4 2.4GHz 2×14-core processor with 128GB of memory within the MESO@LR platform[2]. Each proof attempt was limited to 300 seconds.

8.1 Comparison Between the Variants of Goéland

First of all, we evaluated multiple combinations of Goéland on two problem categories with FOF theorems in the TPTP library: syntactic problems (SYN) and problems of set theory (SET). The former was chosen for its elementary nature, whereas the latter was picked primarily to evaluate the performance of the deduction modulo theory, given that set theory axioms are suitable for rewriting. The comparison involved five variations of Goéland:

[1] https://www.tptp.org/
[2] https://meso-lr.umontpellier.fr/

	SYN (288 problems)		SET (464 problems)	
Goéland	**209** (251 s)		**124** (2 315 s)	
Goéland+EQ	**213** (81 s)	(+6, −2)	**101** (1 585 s)	(+21, −44)
Goéland+DMT	**209** (285 s)	(+0, −0)	**217** (1 294 s)	(+100, −7)
Goéland+DMT +EQ	**213** (119 s)	(+6, −2)	**192** (1 972 s)	(+101, −33)
Goéland+DMT +Polarized	**202** (61 s)	(+1, −7)	**164** (260 s)	(+89, −49)

Table 8.1: Experimental results of the different versions of Goéland over the SYN and SET categories of the TPTP library.

- Goéland
- Goéland+EQ (Goéland improved with equality reasoning)
- Goéland+DMT (Goéland improved with deduction modulo theory)
- Goéland+DMT+EQ
- Goéland+DMT+Polarized (Goéland+DMT improved with polarized deduction modulo theory)

Table 8.1, Figure 8.1 and Figure 8.2 provide detailed results. Table 8.1 shows the number of problems solved by each variant of Goéland, the cumulative time, and the number of problems solved by a given variant but not by the original one (+) and conversely (−). For instance, in the SYN category, Goéland solved 209 problems and Goéland+EQ 213. More exactly, 2 problems were solved by Goéland but not by Goéland+EQ, and 6 problems were solved by Goéland+EQ but not by Goéland. Figure 8.1 and Figure 8.2 present the cumulative time required to solve the number of problems in the two categories.

The results reveal that the variants of Goéland perform relatively similarly in terms of problems solved in the SYN category. This is primarily due to the SYN problems containing a limited number of axioms, which are the triggers for rewrite rules, as well as a few problems with equality. Notably, incorporating equality reasoning leads to a slight increase in the number of problems solved, along with a reduction in the time required.

Specifically, the equality-enhanced versions solve 6 more problems and lose 2. This can be attributed to the fact that some problems that include equality predicates in their axioms may not necessarily need it to be solved. Consequently, equality reasoning may be unnecessarily triggered, leading to some problems being unsolved. The difference in time consumption in the equality version arises from the fact that the unsolved problems (i.e., those that are solved by the basic version) are those that take

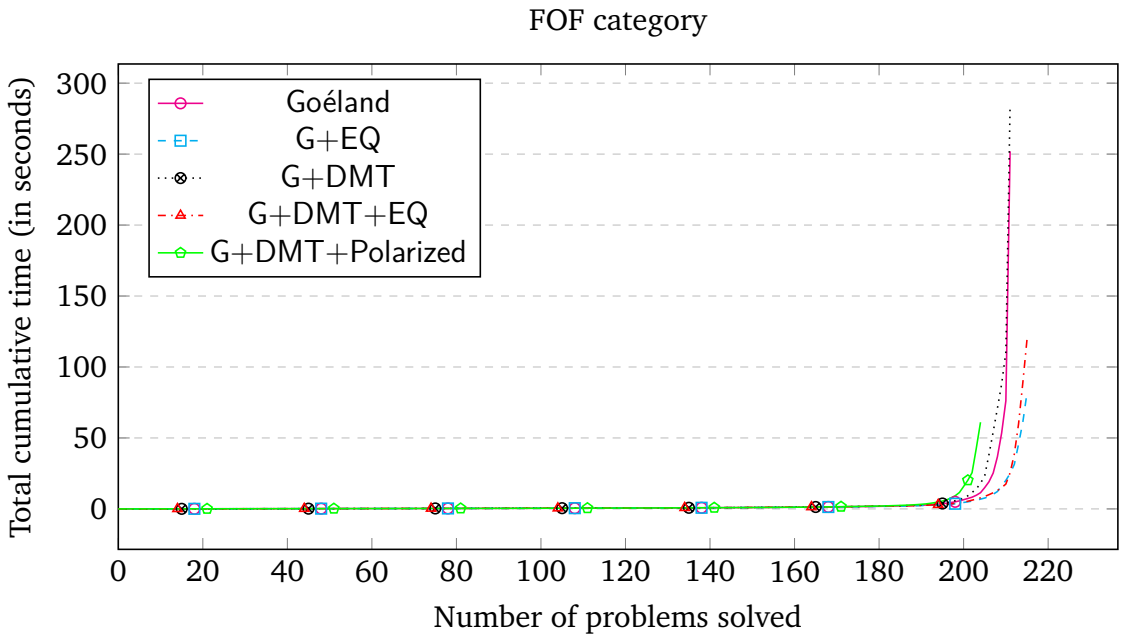


Figure 8.1: Cumulative time per problem solved on the SYN category between Goéland, Goéland+EQ, Goéland+DMT, Goéland+DMT+EQ and Goéland+DMT+Polarized.

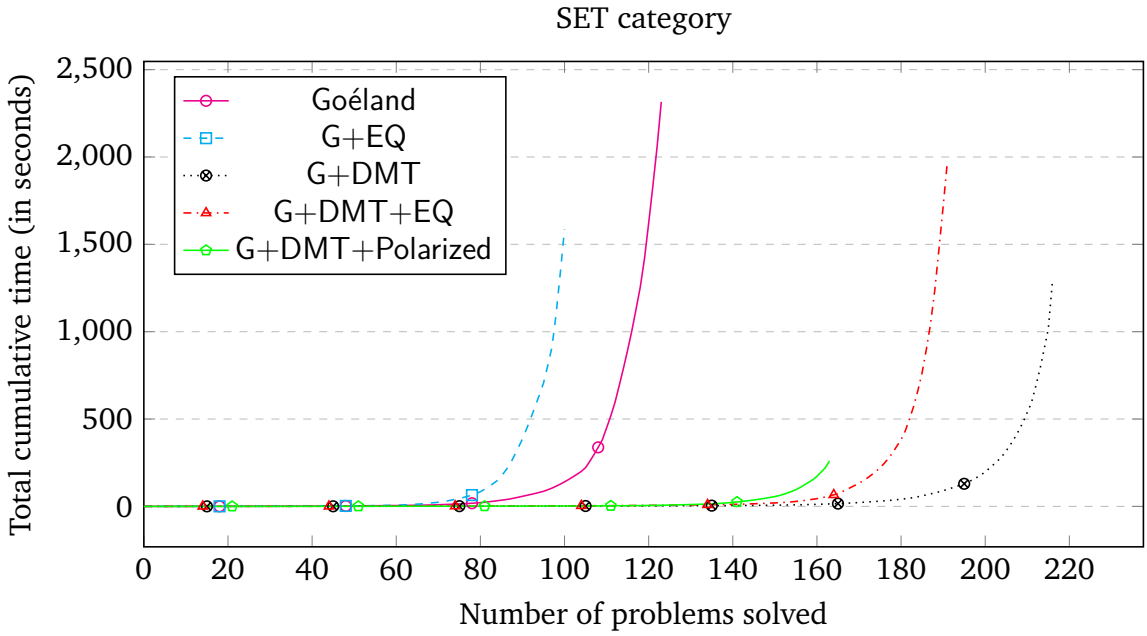


Figure 8.2: Cumulative time per problem solved on the SET category between Goéland, Goéland+EQ, Goéland+DMT, Goéland+DMT+EQ and Goéland+DMT+Polarized.

the most time to be solved, ultimately leading to an overall increase in cumulative time. On the other hand, the addition of deduction modulo also increases the computation time. Indeed, triggering rewrite rules and the associated backtracking takes time, and may not be necessary to solve a problem, resulting in an increase of cumulative time but not in the number of problems solved.

In the SET category, Goéland+DMT demonstrates significantly better results than most other variants, confirming previous findings on the effectiveness of Deduction Modulo Theory for set theory [76, 78]. The polarized version of DMT offers a significant acceleration in terms of proof-search speed, although this gain comes at the cost of lost problems. The versions with equality (resp. DMT+EQ) lead to a drop in the number of problems solved as well as an increase of cumulative time compared to the basic version (resp. DMT version), as equality reasoning introduces additional computational complexity. Furthermore, the equality reasoner is still in its early stages of development and has not yet undergone full optimization, occasionally impacting its efficiency.

For these reasons, for comparison with other theorem provers, we have selected the basic version Goéland, as well as two DMT-improved versions: Goéland+DMT and Goéland+DMT+EQ. This choice will enable us to conduct a more detailed analysis of the improvements offered by DMT, which represent the most advanced version of Goéland.

8.2 Comparison with Other Provers

In order to evaluate the overall performances of Goéland, we have chosen to run it on a larger set of problems and to compare it against various state-of-the-art theorem provers. For this larger-scale experiment, we have chosen a subset of TPTP composed of first-order problems belonging to the following categories: AGT, ALG, COM, CSR, GEO, GRP, HWV, ITP, KLE, KRS, LCL, MGT, MSC, NLP, NUM, PHI, PUZ, SET, SEU, SEV, SWB, SWC, and SYN.

We experimented the three previously chosen versions of Goéland and compared the results with five other provers: tableau-based provers Zenon (v0.8.5) [65], Zenon Modulo (0.4.2 [a5] 2015-09-01) [104] and Princess (v2023-06-19) [211], as well as saturation-based provers Vampire (v4.8) [163] and E (v2.6) [213]. A summary of the total result is available in Table 8.2 and Figure 8.3, and the detailed results for each prover and each category can be found in Appendix B. The tables present the number of problems solved, together with the total cumulative time and the average time. Figure 8.3 follows the same structure as the previous section.

Overall, the results are under most of the other theorem provers. The equality reasoning module is in an early stage of development, and the memory management has not been optimized yet, which leads to difficulties when dealing with very large problems.

However, the DMT-improved versions Goéland+DMT and Goéland+DMT+EQ perform well on some specific categories, for example, SET, GEO (Geometry), or KRS (Knowledge Representation), almost doubling the number of problems solved compared to the basic version. These categories have in common the usage of a

	FOF (5396 problems)
Goéland	**613** (10 482 s — 17.1 s)
Goéland+DMT	**770** (6 935 s — 9 s)
Goéland+DMT+EQ	**801** (10 060 s — 12.5 s)
Zenon	**1 382** (9 026 s — 6.5 s)
Zenon Modulo	**1 389** (10 028 s — 7.2 s)
Princess	**1 621** (23 200 s — 14.3 s)
Vampire	**3 342** (42 873 s — 12.8 s)
E	**3 939** (39 638 s — 10.1 s)

Table 8.2: Experimental results of Goéland, Goéland+DMT, Goéland+DMT+EQ, Zenon, Zenon ModuloPrincess, Vampire and E over a subset of first-order problems of the TPTP library.

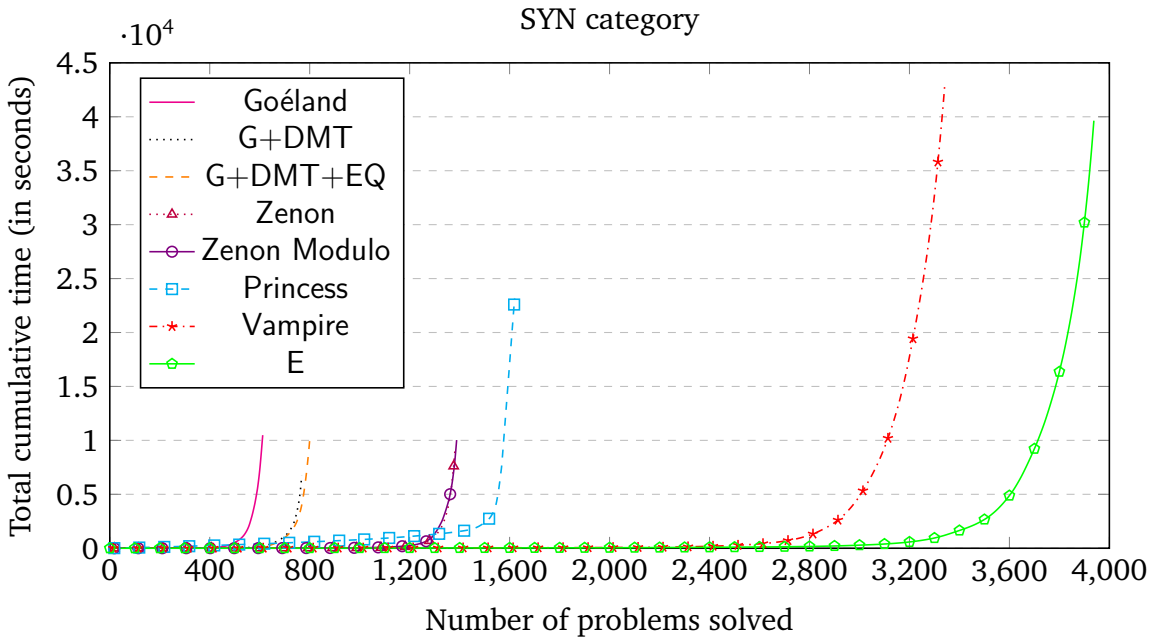


Figure 8.3: Cumulative time per problem solved between Goéland, Goéland+DMT(GDMT), Goéland+DMT+EQ(G+DMT+EQ), Zenon, Zenon Modulo, Princess, Vampire and E.

high number of axioms, making them well-suited for working with deduction modulo theory. These conclusions are confirmed by the increase in the number of problems solved between Zenon and Zenon Modulo on the SET category, whereas the basic version of Zenon already performs well on the two other ones, even without DMT. Deduction modulo theory also increases the performances in terms of speed, dividing by two time needed to solve a problem in Goéland. However, a few problems are also lost in some categories. It can be either due to the non-deterministic proof-search of Goéland, which can provide different search plans for the same input depending on the reception order of the children's answers, or to the application of rewrite rules by the DMT modules, which also modifies the search.

	SYN (207 problems)	SET (113 problems)
2	**1.5 s**	**20 s** (+4)
4	**0.6 s**	**15 s** (+5)
8	**0.4 s**	**12 s** (+8)
16	**0.4 s**	**8.7 s** (+10)
28	**0.3 s** (+ 2)	**8.7 s** (+11)

Table 8.3: Scale-up experimental results of Goéland over the SYN and SET categories of the TPTP library according to the number of cores.

The addition of equality, although leading to a few problems lost in some categories, also improves performances on others that effectively require equality reasoning, such as SWC (software creation).

Even if Goéland manages to achieve results comparable to some other provers on specific categories, it remains overall behind other tableau-based provers, while saturation theorem provers achieve the best results.

8.3 Scale-Up Tests

To assess the impact of the number of CPU cores on Goéland, which is particularly relevant given the concurrent nature of the prover, we have chosen the usual SYN and SET categories of TPTP and the three previous versions of the prover: Goéland, Goéland+DMT and Goéland+DMT+EQ. Tests have been performed on the same machine as the previous one, with an incremental configuration in terms of the number of cores: 2, 4, 8, 16, and 28. We have restricted the problems to those that have been solved for each configuration, i.e., for each number of cores. The mean time is given in seconds and calculated over those common problems. The number of additional problems solved is given in brackets, but the times of these problems are not included in the mean.

A summary of the total results is available in Table 8.3, illustrated in Figure 8.4 and 8.5 for Goéland and in Table 8.4, illustrated in Figure 8.6 and 8.7 for Goéland+DMT. For example, for Goéland on the SYN categories (Table 8.3), the same 207 problems have been solved regardless of the number of cores, and two additional problems have been solved on 28 cores.

First of all, we can observe that increasing the number of cores generally makes the average time decrease, leading to an improvement in performance. The stagnation and the variation in terms of problems solved can be explained by the non-deterministic nature of Goéland since executing the prover on one problem can lead to different (and possibly longer or unsuccessful) proof searches. In addition, the increase of cores comes with an increase in the number of goroutines active at the same time and the need to manage them, which relies on the scheduler. Thus, the time gained by the parallelization may be compensated by the time taken to switch between the processes. This can explain the stagnation despite the increase of cores. The number of problems

	SYN (207 problems)	SET (208 problems)
2	**1.4 s** (+ 1)	**6.1 s** (+ 5)
4	**1.3 s**	**5.3 s** (+ 8)
8	**1.1 s**	**4.7 s** (+ 7)
16	**0.6 s** (+ 1)	**4.2 s** (+ 9)
28	**0.4 s** (+ 2)	**3.1 s** (+ 9)

Table 8.4: Scale-up experimental results of Goéland+DMT over the SYN and SET categories of the TPTP library according to the number of cores.

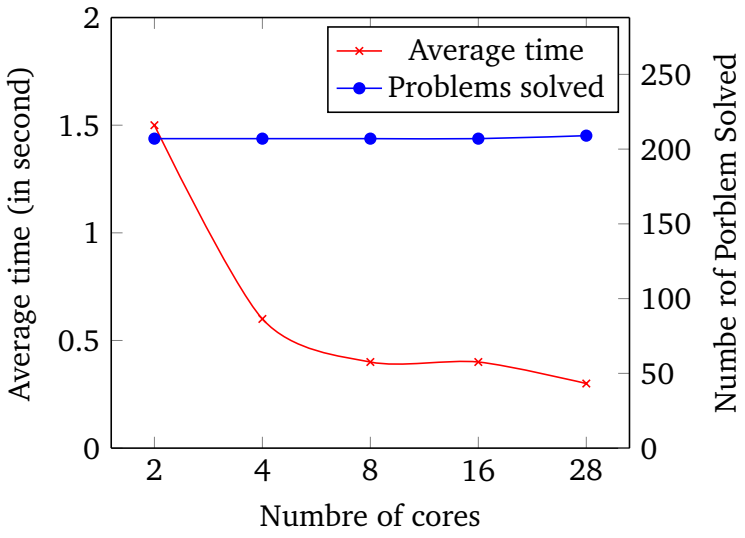


Figure 8.4: Scale-up results of Goéland on the SYN category.

solved also follows this trend, with a slight increase or stagnation, although some problems can be lost due the the reasons aforementioned.

8.4 Typed Problems

We also conducted experiments on the polymorphism module of Goéland. For these tests, we selected TFF problems from TPTP that do not contain arithmetic, as Goéland does not currently have a proper extension to handle it. The TFF category is thus reduced to problems from COM (31), HWW (68), ITP (342), LCL (66), NUM (49), PUZ (10), SCT (66) and others (GEO, KRS and SYN, one each). We compared two versions of Goéland+DMT (with and without equality reasoning) against two versions of Zenon (the basic version and Zenon Modulo). The results of these experiments can be found in Table 8.5.

Despite a relatively small number of problems solved, the results demonstrate that Goéland is capable of handling typed problems. The results are proportionally comparable to those achieved in the FOF category, indicating that the challenges faced are similar in both categories. On the other hand, Zenon successfully solves

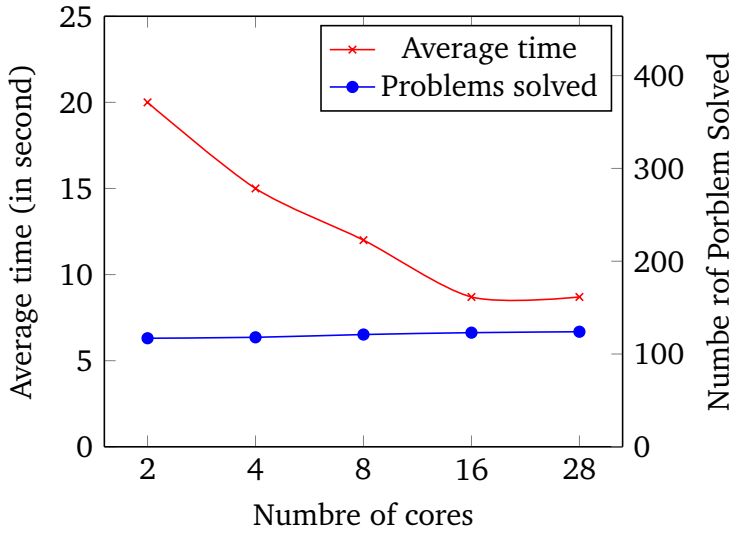


Figure 8.5: Scale-up results of Goéland on the SET category.

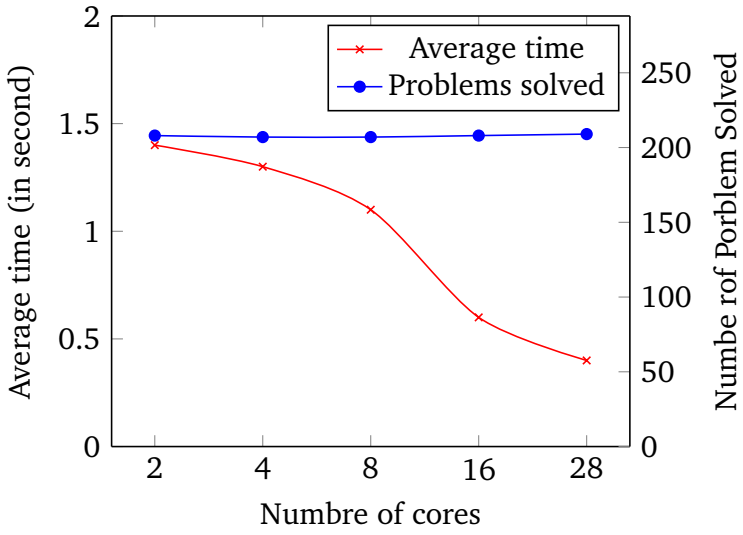


Figure 8.6: Scale-up results of Goéland+DMT on the SYN category.

a significant number of problems, though the DMT extension appears to be less efficient than the original version.

Notably, in the primary categories within TFF, either both Zenon and Zenon Modulo yield results comparable to the corresponding FOF categories, either the latter is experiencing a drop in the number of problems solved, particularly in NUM and LCL categories. One possible explanation for this lies in the fact that in these categories, numerous rewrite rules can be triggered by the same axiom, compelling the deduction modulo theory mechanism to choose between them, potentially not selecting the one leading to a solution.

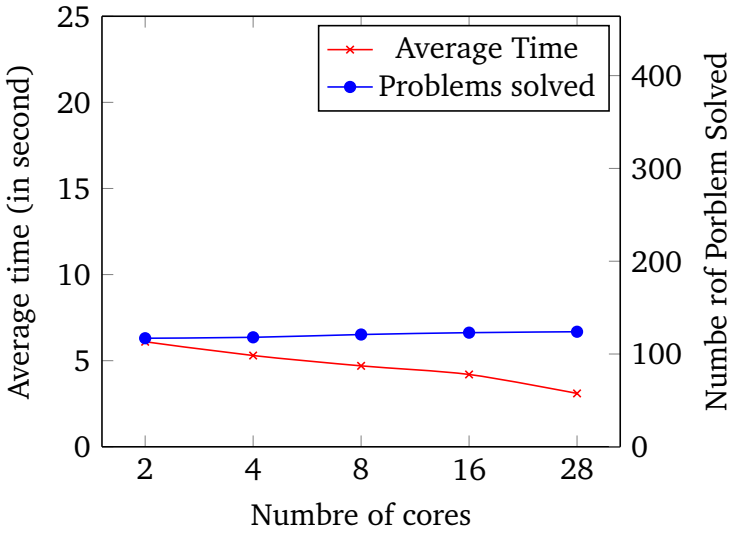


Figure 8.7: Scale-up results of Goéland+DMT on the SET category.

	TFF (without Arithmethic) (635 problems)
Goéland+DMT	**38** (10.7 s)
Goéland+DMT+EQ	**27** (3.1 s)
Zenon	**116** (1.7 s)
Zenon Modulo	**87** (1.2 s)

Table 8.5: Experimental results of Goéland+DMT, Goéland+DMT+EQ, Zenon and Zenon Modulo over a subset of the TFF problems of the TPTP library.

8.5 Expansion of the Proof Size with Deskolemization Strategy

In order to evaluate the proof certification algorithm, we used the SYN and SET categories of TPTP. To refine the set of selected problems, we launched Goéland and Goéland+DMT (with all types of Skolemization) on the two categories to create a subset of the problems proved by at least one variant. These subsets, together with Goéland and the benchmark script, are available online[3].

Then, the tests have been launched on the six following variants: Goéland, Goéland+δ^+, Goéland+δ^{+^+}, Goéland+DMT, Goéland+DMT+δ^+ and Goéland+DMT+δ^{+^+}. Each variant corresponds to a particular Goéland's option set, where the δ^+-rules can be activated using the `-inner` flag, δ^{+^+}-rules with the `-preinner` flag and DMT with the `-dmt` flag. Each benchmark generates a folder containing, for every problem proved (i) its tableau proof and (ii) its Coq proof, certified by Coq's compiler. Then, the statistics for a test can be generated using a script, also available online. It is important to note that, as Goéland is a parallel theorem prover, its proof search algorithm is *non-*

[3]Benchmarked problems available on https://github.com/GoelandProver/GoelandBenchmarks/ in the `PROOF_CERTIFICATION` folder.

	Problems Proved	Percentage Certified	Avg. Size Increase	Max. Size Increase
Goéland	261	100 %	0 %	-
Goéland+δ^+	272	100 %	8.1 %	5.3
Goéland+δ^{+^+}	274	100 %	10.6 %	10.3
Goéland+DMT	363	100 %	0 %	-
Goéland+DMT+δ^+	375	100 %	4.5 %	3.9
Goéland+DMT+δ^{+^+}	377	100 %	7.4 %	5.2

Table 8.6: Comparison between the different Skolemization strategies and their proof-size increase.

deterministic and as such, the results presented here may not be perfectly reproducible. Nevertheless, in any case, the percentage of problems certified should be near 100%.

Table 8.6 presents an overview of the results for the previously explained benchmarks. The first column contains the name of the variant that corresponds to the row's results. The second shows the number of problems on which a *tableau proof* has been output by the variant. The third column gives the percentage of tableau proofs that have been successfully translated to Coq's proofs. The fourth and fifth columns present the size increase between the tableau proof and the Coq's proof, in terms of the number of branches. The former exhibits the average size increase between both proofs while the latter indicates the maximum ratio obtained in the considered subset.

The results obtained are very promising as (i) every proof of all the variants has been properly certified and (ii) the average size increase between the two versions of the proof is low. In theory, a tableau proof is exponentially better than its GS3 counterpart in inner Skolemization. However, for both variants featuring this Skolemization strategy, the average and maximum increase of size is low. Indeed, for the Goéland+δ^+ variant, on average, only two more branches are created during the translation. Furthermore, the maximum increase is realized on a 48-branches proof (of SYN867+1). The translation features 255 branches, i.e., $2^8 - 1$ branches, which is 2^{40} times less than the theoretical bound. The variant Goéland+DMT+δ^+ is even better, as on average it does not even increase the proof size of one branch, and the maximum ratio is reached for the same problem as the former variant, yielding a 153-branches proof from a 39-branches one. For δ^{+^+}-variants, as expected, the increase is more pronounced. It is still, however, a relatively low increase in proof size, as this Skolemization strategy is theoretically exponentially better compared to δ^+-rules.

The average increase in non-DMT mode consists of more than two and a half branches while the maximum ratio (still on the same problem) is reached on a 34-branches proof, with the translation yielding 352 branches, which is also a long way from the theoretical bound. Meanwhile, the DMT variant is even better, with an average size increase of a little more than one branch with the maximum ratio being attained by a 25-branches proof. It is not necessary to expand on variants with an outer Skolemization strategy, as it behaves as expected: a one-to-one translation

between the tableau proof and the Coq proof is realized, easily certifying everything with the exact same proof size.

All in all, the results obtained are more than satisfactory as they validate the translation algorithm by yielding relatively short proofs. It means that it is cheap, in practice, to certify proofs using the proposed translation algorithm together with an embedding in a proof assistant.

8.6 Conclusion

Despite a relatively small number of proven problems, considering the early stage of development of the prover, the results are promising, particularly with the deduction modulo theory.

Benefits provided by the equality reasoner are minimal due to its lack of efficiency. To enhance its performance, we can consider adding more restrictions on the $\mathcal{BSE}$ calculus, revisiting the level of parallelization, or entirely revamping the equality reasoning mechanism. In a broader context, improving the memory management of Goéland is also essential to handle larger problems.

However, deduction modulo theory stands out as a promising optimization. It offers improvements comparable to those achieved with other DMT tools, validating prior work with this method. While its extension to polarized deduction modulo theory shows an interesting increase in speed, one of the upcoming challenges is to find a good balance in terms of the number of rewrite rules, to retain only the relevant ones and apply them judiciously, without the risk of being overwhelmed by excessive rewrite steps.

Regarding the output of checkable proofs, the promising results showed in practice by the algorithm over δ^{+}-rules proofs have then comforted us in implementing the extensions to handle $\delta^{+^{+}}$-rules and deduction modulo theory, two other proof optimization techniques. Empirically, these extensions also yield promising results, thus showing an alluring future work path in the lifting of the translation towards the other forms of Skolemization: δ^{ε}-, δ^{*}- and $\delta^{*^{*}}$-rules.

In conclusion, while some experiments with deduction modulo theory or checkable proofs have yielded interesting results, there is ample room for technical improvement to make the tool more competitive.

Conclusion

The idea developed throughout this thesis was to extend the field of first-order automated reasoning through the study of the method analytic tableaux, by combining it with current techniques. In detail, we studied the use of concurrency for the development of a tableau-based proof-search procedure with eager closure, its interactions with some theory reasoning techniques, and the transformation of the generated proofs in order to be checked by external tools. Our main contribution relies on the creation Goéland theorem prover, which incorporates most of our theoretical results.

Contributions

Fairness in Tableau-Based Proof Search

The proof-search procedure of Goéland, which is based on tableaux, tackles most of the fairness challenges thanks to its concurrent branch exploration and its forbidden substitutions mechanism. In detail, each branch performs its own proof search, and the final decision relies on an agreement mechanism, in which each potential solution is tried before being validated or forbidden.

This strategy has demonstrated effectiveness in addressing most of the fairness challenges inherent to tableaux. Moreover, it has been proven complete, increasing the confidence in the method and contributing to the growing body of completeness proofs for proof-search methods in automated deduction.

Theory Reasoning in First-Order Logic

In order to make Goéland applicable in highly specialized domains, and as the management of theories in tableaux is not uniform, we studied the incorporation of two different types of background reasoners within a tableau-based proof-search procedure. Thus, two background reasoning modules were developed: one for handling equality reasoning using rigid E-unification, and another to deal with any axiomatized theory with deduction modulo theory.

We placed particular emphasis on concurrency, given its dependency on the theory it addresses, by highlighting aspects amenable to parallelization and their critical interactions with the proof-search process. The equality reasoner, for instance, leverages concurrency to attempt to find a solution as soon as possible, while parallelization in the deduction modulo theory module is primarily applied to the search for applicable rewrite rules.

Proof Certification

Finally, in order to reinforce integration with proof assistants, we explored methods for generating checkable proofs, even with more advanced Skolemization strategies.

The deskolemization mechanism, by ensuring equivalence between proofs in different formats, enables one to carry out an efficient proof search with the use of optimized Skolemization strategies while preserving the fundamental benefits of tableau methods: the production of a proof. This transformation essentially makes the automated theorem prover evolve into a proof-certificate generator, with the generated certificates subsequently subject to validation by an external proof checker.

This approach does not only allow to improve performance of Goéland, but also maintains its compatibility with software that operates on unaltered formulas. Thus, two extensions have been developed to export those proofs into Coq and Lambdapi, making them machine-checkable.

Perspectives

Although the practical results cannot yet compete against other state-of-the-art provers, they remain promising, especially in some domains thanks to the use of deduction modulo theory. However, several areas for potential improvement can be envisioned.

Fairness in Tableau-Based Proof Search and Practical Results

The theoretical aspects of fairness management in the procedure are noteworthy as they address known challenges in this domain. To make Goéland suitable for real-world applications, future efforts will focus on strengthening the prover's foundation, including a comparison with a memory-shared version. Notably, it is also essential to rework memory management, as the prover currently encounters difficulties with large problem files. We also need to conduct a more comprehensive failure analysis, as Goéland is unable to find a proof for some problems that are typically considered "easy". While in some cases it can find a proof using more advanced Skolemization strategies or by attempting a different substitution earlier in the proof search, there are still problems that inherently pose a significant challenge for the prover.

Moreover, the use of heuristics can enhance the proof search process. Presently, when dealing with a substantial number of axioms that cannot be translated into rewrite rules, they are processed in an arbitrary order, lacking specific inclination. Thus, one can imagine implementing a heuristic that associates a priority score to each formula to process, related to its proximity to the conjecture. Similar reasoning can be applied to the choice of substitution. Currently, the prover selects an arbitrary substitution among those with the fewest non-local free variables, but this process could be improved by considering the reintroduction degree of each variable.

These optimizations can also involve a reconsideration of the eager closure approach. One possibility is to sacrifice tool completeness by only considering the initial answers from each branch. This may increase the substitution failure rate and, consequently, lead to more frequent backtracking, but it has the potential to yield a solution more rapidly. On the other hand, an alternative approach is to explore delaying the closure rule, in line with the incremental closure concept proposed by Giese [136]. This strategy aims to reduce the number of backtracks by attempting potential closing substitutions at each proof-search step while allowing branches to

continue extending. Tests must be performed for those two approaches, for which implementations are currently in progress.

Finally, one of the primary challenges in automated reasoning is its inherent automation, which renders it relatively static and devoid of human intuition. Incorporating machine learning methods could potentially infuse a level of "human intuition", suggesting the right substitution to try, providing a more precise γ-rule application limit, or guiding the selection of a useful formula to process in the next proof-search step. This practice has shown promise in some automated theorem provers and SMT solvers [155, 240].

Theory Reasoning in First-Order Logic

Addressing the efficiency of the equality reasoning module is a priority for the theory reasoning of Goéland. This can involve optimization of the current implementation or considering alternative approaches, such as a current work focused on the addition of equality rules directly integrated into the tableau calculus. Additionally, we can study the interactions between deduction modulo theory and equality, especially to integrate term rewriting. Indeed, as both of the mechanisms work with the same set of terms, their interactions could be challenging.

Moreover, while Goeland can handle typed problems thanks to its native encoding of polymorphic types, it currently does not benefit from any specific procedure for typed theories, such as arithmetic, for which the development of a dedicated reasoner based on a simplex and branch and bound approach reasoner is in progress. Additionally, to expand the utility of Goéland in program verification, extending the management of Typed First-order Logic (TFF) to Extended Typed First-Order Form (TXF) could be considered. TXF includes support for tuples, conditional expressions (if-then-else), and let expressions (let-defn-in).

In spite of its promising results, deduction modulo theory can be further enhanced. Improvements can be made by manually designing rewrite rules, either independently or in conjunction with automated computation from axioms. For instance, the BWare project, which is based on set theory, provides manually designed rewrite rules for reasoning about industrial problems. Further testing on this benchmark, with the addition of arithmetic reasoning, can offer valuable insights

Additionally, while polarized deduction modulo theory demonstrates impressive speed, it comes with a loss of problems. Heuristics can be developed in order to achieve the right balance in computing rewrite rules and benefit from the speed increase without sacrificing problems.

Proof Certification

Since being able to produce verified proofs is valuable, there is a desire to extend the output to other proof assistants, broadening Goéland's verification capabilities and enhancing interoperability.

Furthermore, the true question revolves around the placement of deskolemization process: should it be directly implemented into the prover or integrated into the proof

checker? In the former case, we can envision the creation of a standardized format for the output of tableau proofs, eliminating the need for individually incorporating a deskolemization process into each automated theorem prover. These proofs could then be processed by a specialized tool that implements the translation algorithm within a certified environment, simplifying the process of automatically certifying tableau proofs.

On the other hand, an alternative approach involves implementing advanced Skolemization strategies directly within the format itself, shifting the responsibility to the proof checker. In this scenario, deskolemization becomes unnecessary, as proofs can be directly translated into the corresponding proof assistant. This approach also raises the question of the quality of the certificate. It is preferable for a proof to be concise, omitting computational steps, which are left to the proof checker [178]. However, integrating such a mechanism into a proof checker may pose challenges, and it does not eliminate the need for a common output format that serves as the basis for developing common translation rules for specific proof assistants.

Résumé de la thèse

Avec l'augmentation de la prévalence et de la complexité des systèmes informatiques, leur fiabilité est devenue une préoccupation cruciale, en particulier dans le cadre des systèmes critiques. Tout bogue ou dysfonctionnement dans ces systèmes peut avoir de graves conséquences, à la fois en termes financiers et, plus important encore, en termes humains. Parmi les exemples notables de bogues, il est possible de citer la défaillance de la fusée Ariane 5 de l'Agence spatiale européenne et le bogue du Pentium FDIV d'Intel.

Si le *test* est une méthode courante pour détecter et atténuer les bogues d'un système, il n'est pas exhaustif et ne peut garantir l'absence totale de défauts. À l'inverse, les *méthodes formelles* permettent de démontrer, à l'aide d'un raisonnement mathématique rigoureux, qu'un système fonctionne exactement comme prévu, sans aucune déviation. Bien qu'elles nécessitent davantage de moyens, tant financiers qu'humains, les méthodes formelles restent le seul moyen de garantir l'exactitude d'un système dans l'ensemble de son espace de fonctionnement.

Cette évaluation repose sur l'existence d'une *preuve* pour un problème donné. Une preuve est une séquence de déductions visant à exhiber le mécanisme de raisonnement inhérent, conduisant à la validation de la propriété initiale. Historiquement, les preuves étaient réalisées par un être humain à l'aide d'un papier et d'un crayon et étaient l'apanage d'une poignée de spécialistes. Cependant, au fil du temps, de nouvelles techniques sont apparues et les preuves peuvent désormais être établies en collaboration avec des ordinateurs, en utilisant différents degrés d'automatisation. Ces niveaux d'automatisation varient des *assistants de preuve*, qui guident les utilisateurs humains dans la construction d'une preuve tout en garantissant des dérivations sans erreur, aux *prouveurs automatiques de théorèmes*, qui génèrent des preuves de manière indépendante et algorithmique.

Les *outils de raisonnement interactif* agissent comme des assistants, guidant les utilisateurs humains dans la construction de preuves tout en garantissant l'exactitude des dérivations générées. Le raisonnement interactif joue un rôle important en mathématiques, aidant les mathématiciens à prouver des théorèmes et à vérifier des composants essentiels dans des systèmes ayant des implications critiques pour la sécurité humaine. Par exemple, le système d'exploitation `seL4` a été vérifié à l'aide de l'assistant de preuve `Isabelle/HOL`, et l'assistant de preuve `HOL Light` a été un outil clé dans la résolution de la conjecture de Kepler en 2017.

Les *outils de raisonnement automatiques* sont capables de raisonner entièrement ou partiellement automatiquement sur des formules logiques. Ces outils de raisonnement

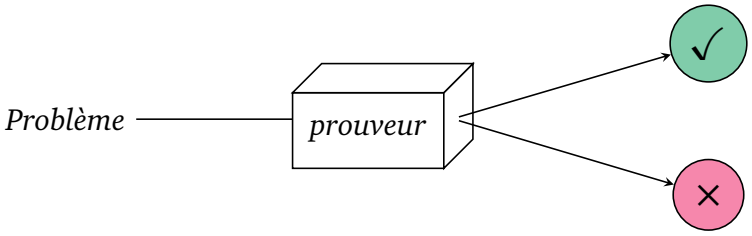


Figure 9.8: Un *prouveur* prend un problème et retourne une information à propos de sa valeur de vérité.

sont également appelés *prouveurs* (Figure 9.8), et sont largement utilisés dans des domaines tels que la vérification et le test de programmes, l'ordonnancement, ainsi que pour résoudre des problèmes mathématiques. À titre d'illustration, la *méthode B* a été utilisée pour vérifier la fonctionnalité de la ligne 14 du métro parisien. Les outils de raisonnement interactifs et automatiques peuvent être utilisés conjointement, par exemple en confiant les parties les plus simples d'une preuve construite de manière interactive à des prouveurs automatiques de théorèmes.

Bien que les preuves nous permettent d'atteindre un degré de confiance satisfaisant dans nos systèmes, la complexité des outils permettant de les générer a historiquement confiné leur utilisation à une minorité de spécialistes. Cependant, les récents développements en matière de déduction automatique, les capacités de calcul accrues et les efforts visant à améliorer l'accessibilité des logiciels de preuve ont démocratisé ces techniques auprès d'un plus large spectre d'utilisateurs. Bien qu'il soit aujourd'hui plus facile que jamais de collaborer avec des ordinateurs pour construire des preuves, il reste nécessaire d'améliorer l'ergonomie pour les utilisateurs finaux et de développer des outils automatiques capables de produire des preuves certifiées. Ces développements permettront une application plus large des méthodes formelles dans la vérification des programmes, contribuant ainsi à la création de systèmes logiciels plus sûrs.

Cette thèse porte sur la conception et le développement d'un prouveur automatique de théorèmes. De tels outils reposent sur deux caractéristiques principales : ce sur quoi ils raisonnent et comment ils le font. Le premier aspect concerne le choix du langage utilisé pour décrire les problèmes, tandis que le second se réfère aux techniques de raisonnement.

En effet, les prouveurs automatiques de théorèmes ne peuvent pas s'attaquer directement aux logiciels ou aux problèmes mathématiques, et nécessitent d'abord une traduction de ces problèmes du monde réel dans un langage commun lisible par les ordinateurs, qui permet aux différents outils de communiquer entre eux. Ce langage commun est la *logique* et permet de formaliser le monde. En raison du large

éventail et de la diversité des logiques existantes, une myriade de concepts peut être représentée, depuis les concepts mathématiques jusqu'à de complexes problèmes industriels. Ces logiques se différencient en fonction de leur niveau d'expressivité et de l'efficacité des méthodes de raisonnement associées.

Certaines logiques, telles que la logique propositionnelle, sont décidables et disposent de techniques particulièrement efficaces. En contrepartie, leur degré d'expressivité est limité, et seuls des concepts simples peuvent être représentés. À l'inverse, les logiques d'ordre supérieur offrent une plus grande expressivité, mais posent souvent problème aux outils de raisonnement automatique, notamment pour parvenir à un raisonnement efficace. Dans le contexte de l'automatisation, la logique du premier ordre se présente comme un bon candidat, offrant un équilibre intéressant entre expressivité et efficacité de raisonnement. D'une part, elle permet la représentation des individus et des propositions les concernant. D'autre part, bien que semi-décidable, elle peut se targuer de techniques de raisonnement efficaces, facilitant les déductions à partir de problèmes réels et mathématiques, également appelés *formules logiques*.

La sélection d'un langage influence considérablement le choix des techniques de raisonnement employées. En effet, les différentes techniques ne sont pas applicables à toutes les logiques et dépendent de facteurs tels que la transformation de la formule initiale ou le résultat souhaité. Ces techniques de raisonnement sont appelées *procédures de recherche de preuves*. Comme leur nom l'indique, elles visent à rechercher une preuve en appliquant un ensemble de règles sur une version potentiellement modifiée des formules initiales, en explorant l'espace de recherche de preuves.

Il existe une multitude de techniques de raisonnement automatique, chacune ayant ses propres caractéristiques. Certaines de ces techniques sont connues pour leur efficacité, tandis que d'autres possèdent des propriétés qui peuvent être avantageuses dans un certain contexte, comme une entrée spécifique ou l'adaptation à un spectre plus large de logiques.

L'une de ces techniques se nomme *la méthode des tableaux analytiques*. Cette méthode fonctionne de manière syntaxique, en déconstruisant la formule initiale à prouver en sous-formules, jusqu'à atteindre des axiomes dont la véracité est prouvée. Notamment, cette méthode fonctionne avec la formule initiale, sans transformation, ce qui la rend à la fois adaptée aux interactions avec les assistants de preuve et utilisable dans d'autres types de logiques. Dans le contexte de la logique du premier ordre, sa principale force réside dans sa capacité à produire une preuve, qui peut être facilement traduite en une preuve vérifiable par un outil externe.

Défis

Équité dans la recherche de preuve basée sur les tableaux

La méthode des tableaux analytiques se heurte cependant à certaines difficultés qui peuvent l'empêcher de trouver une preuve. La structure arborescente générée par la décomposition des formules originales en sous-buts peut introduire des dépendances entre les branches, ce qui accroît la complexité de la recherche de preuves. De plus, la variété des choix disponibles à chaque étape de la preuve rend le système sujet à des problèmes d'équité, comme l'a déclaré Hähnle : « À l'heure actuelle, on ne connaît pas de procédure de technique de preuve utilisant les tableaux destructifs fortement complète qui fonctionne bien en pratique » [208].

De plus, la plupart des livres décrivent la *fermeture anticipée* comme la manière standard de gérer la fermeture dans les tableaux à variables libres. En plus du problème d'équité qui peut être induit par cette règle, il reste difficile de prouver la complétude d'une procédure de recherche de preuves avec fermeture anticipée, car elle implique un méchanisme de retour sur trace et donc une non-monotonicité. Cependant, la complétude est un défi critique pour toute procédure de recherche de preuves, car un outil complet permet d'instaurer la confiance dans ses résultats. Dans le contexte des preuves de complétude standard pour les procédures basées sur des tableaux du premier ordre, ces dernières impliquent souvent de considérer l'arbre de preuve (infini) qui résulterait si la procédure ne parvenait pas à se terminer sur une formule insatisfaisante, conduisant finalement à un contre-modèle et à une réfutation. Cependant, pour les procédures de recherche de preuves avec retour sur trace, la construction de cette dérivation infinie n'est pas simple. Il est donc difficile d'obtenir une procédure de recherche de preuve équitable, complète et efficace dans les tableaux.

Raisonnement au sein de théories en logique du premier ordre

Au-delà de la procédure de recherche de preuves elle-même, certains problèmes sont intrinsèquement difficiles ou exigent des approches adaptées à des contextes spécifiques. Ceci est particulièrement présent dans les applications industrielles, qui impliquent souvent des contraintes explicites ou des structures de données telles que les tableaux ou les tas, ou dans les théorèmes mathématiques qui se rapportent à des théories spécifiques, comme la théorie des ensembles. Ces problèmes impliquent un large ensemble d'axiomes, qui fournissent le contexte nécessaire pour prouver la formule, ainsi qu'une sémantique spécifique ou des techniques de raisonnement dédiées, telles que celles pour l'égalité ou le raisonnement arithmétique. Avec

l'augmentation de la complexité des systèmes, la capacité à traiter les théories est une préoccupation cruciale pour tout prouveur automatique de théorèmes contemporain.

Dans le contexte de la logique du premier ordre, le raisonnement théorique est difficile, mais néanmoins essentiel. Bien qu'il existe des méthodes efficaces pour aborder des domaines spécifiques, il n'existe pas d'approche unique pour traiter toutes les théories possibles. En outre, l'inclusion des axiomes des théories dans les hypothèses des problèmes est rarement utilisable en pratique dans les scénarios du monde réel, car elle implique souvent une application irréfléchie des axiomes qui surcharge le processus de recherche de preuves. Toutefois, des stratégies ont vu le jour pour relever ces défis. En particulier, bien que principalement axé sur des théories spécifiques, *déduction modulo théorie* a évolué et peut être utilisé comme une optimisation pour les prouveurs automatiques de théorèmes. En transformant les axiomes en règles de réécriture, il permet de ne déclencher que les règles pertinentes, ce qui réduit l'espace de recherche et rend la recherche de preuves plus efficace. Néanmoins, son intégration dans une recherche de preuve basée sur un tableau n'est pas simple, car elle interagit étroitement avec des mécanismes critiques tels que la dépendance des variables libres.

Certification de preuves

D'une certaine manière, les prouveurs automatiques de théorèmes peuvent être considérés comme des oracles, générant une réponse pour une formule donnée. Si certains d'entre eux tentent de fournir une trace, ils peuvent également se limiter à la production d'une réponse binaire (oui ou non) dans le pire des cas. La fiabilité de la réponse dépend alors uniquement du niveau de confiance que nous avons dans le prouveur concerné. Néanmoins, ces outils sont généralement des logiciels de grande envergure, comprenant des dizaines de milliers de lignes de code et employant des heuristiques sophistiquées. En outre, étant développés par des humains, ils sont susceptibles d'être affectés par des bogues et intrinsèquement sujets à des erreurs. Dans de tels outils, les bogues peuvent être désastreux, les amenant à prouver des non-théorèmes et, par conséquent, à compromettre la fiabilité des réponses qu'ils produisent. Heureusement, il existe deux façons d'éviter les incohérences dans les prouveurs automatiques de théorèmes : certifier entièrement le noyau du prouveur à l'aide d'un assistant de preuve, ce qui est un travail coûteux, ardu et de longue haleine [212], ou produire des preuves vérifiables par un outil externe, ce qui est généralement facile d'accès.

Ce dernier s'appuie sur la notion de certificats de preuve. Il s'agit de preuves générées par un prouveur de théorèmes automatisé qui peuvent être vérifiées par un

vérificateur externe. En effet, à l'inverse des prouveurs automatiques, les assistants de preuve s'appuient sur un noyau certifié, garantissant l'exactitude des preuves vérifiées. Il est donc naturel de chercher un moyen de combiner les forces des deux mondes en produisant des preuves vérifiables, inspirant ainsi une confiance totale dans les résultats du prouveur automatique. En outre, le fait de s'appuyer sur un vérificateur de preuves externe pour valider les preuves renforce considérablement la confiance que nous leur accordons, tout en établissant un cadre commun pour l'expression des preuves. L'un des avantages de ce cadre commun est la possibilité d'échanger des preuves provenant de divers prouveurs de théorèmes qui peuvent utiliser des systèmes de preuve différents. Cependant, toutes les méthodes de raisonnement du premier ordre ne peuvent pas être facilement traduites en preuves vérifiables, en particulier lorsque des heuristiques avancées sont utilisées.

Contributions

Cette thèse vise à relever un large éventail de défis, avec l'ambition de faire progresser le domaine de la déduction automatique en utilisant les tableaux analytiques au premier ordre. Les principales contributions englobent à la fois des aspects théoriques et pratiques, ces derniers conduisant à des développements qui mettent en œuvre nos résultats théoriques dans un nouvel outil appelé Goéland. Cet outil comprend les éléments clés suivants :

- Une procédure de recherche de preuve concurrente basée sur des tableaux qui garantit l'équité par construction. En effet, la structure arborescente offerte par cette méthode s'adapte bien à un traitement concurrent. En concevant une procédure qui explore les branches en parallèle et en tirant parti des informations d'une branche pour éliminer plus rapidement certains sous-espaces de recherche, cette thèse vise à résoudre les problèmes d'équité au sein de la méthode de tableaux. Cette procédure a été prouvée complète, ce qui constitue, à notre connaissance, la première preuve de complétude d'une procédure basée sur la méthode des tableaux analytiques en logique du premier ordre avec fermeture anticipée.

- L'implémentation de deux raisonneurs de fond pour traiter les théories : un raisonneur pour gérer le traitement de l'égalité et un module de déduction modulo théorie. Dans cette thèse, nous approfondissons l'incorporation d'un module de raisonnement dédié et d'un autre plus général, en examinant leurs interactions respectives avec une procédure de recherche de preuve concurrente basée sur des tableaux.

- Une procédure de traduction des preuves par tableaux vers une structure de preuve générique vérifiable par un outil externe, à savoir GS3, ainsi que deux sorties vers des assistants de preuve dédiés : Coq et Lambdapi.

Nos principales contributions, en plus du travail de mise en œuvre pure, sont multiples et, par conséquent, détaillées dans des chapitres distincts. Le manuscrit est donc organisé comme suit. Le chapitre 1 introduit des notions préliminaires de logique et de concurrence, tandis que l'état de l'art en la matière est disponible dans le chapitre 2. La procédure principale, ainsi que les défis à relever, sont présentés dans le chapitre 3 et est prouvée complète dans le chapitre 4.

Afin d'étendre les possibilités de Goéland, le chapitre 5 présente le raisonnement théorique mis en œuvre dans le prouveur et le 6 son implémentation, ainsi que celle d'autres fonctionnalités annxes. Pour finir, le chapitre 7 introduit une stratégie pour produire des preuves vérifiables, qui est testé, ainsi que toutes les fonctionnalité de Goéland, sur la bibliothèque TPTP dans le 8.

L'idée développée tout au long de cette thèse est l'extension du domaine du raisonnement automatique au premier ordre au travers de l'étude de la méthode des tableaux analytiques, en la combinant avec les techniques actuelles. En détail, nous avons étudié l'utilisation de la concurrence pour le développement d'une procédure de recherche de preuve basée sur des tableaux avec fermeture anticipée, ses interactions avec certaines techniques de raisonnement théorique, et la transformation des preuves générées afin qu'elles puissent être vérifiées par des outils externes. Notre principale contribution repose sur la création du prouveur Goéland, qui intègre la plupart de nos résultats théoriques.

Retour sur les contributions

Équité dans la recherche de preuves basée sur les tableaux

La procédure de recherche de preuves de Goéland, qui est basée sur des tableaux, résout la plupart des problèmes d'équité grâce à son exploration simultanée des branches et à son mécanisme de substitutions interdites. Dans le détail, chaque branche effectue sa propre recherche de preuve, et la décision finale repose sur un mécanisme d'accord, dans lequel chaque solution potentielle est essayée avant d'être validée ou interdite.

Cette stratégie s'est révélée efficace pour résoudre la plupart des problèmes d'équité inhérents aux tableaux. De plus, cette procédure a été prouvée complète, ce qui augmente la confiance dans ses résultats et contribue au nombre croissant de preuves de complétude pour les outils de démonstration de théorèmes.

Raisonnement au sein de théories en logique du premier ordre

Afin de rendre Goéland utilisable dans des domaines hautement spécialisés, et étant donné que la gestion des théories dans les tableaux n'est pas uniforme, nous avons étudié l'incorporation de deux différents types de moteurs de raisonnement de fond au sein d'une procédure de recherche de preuve basée sur des tableaux. Ainsi, deux modules de raisonnement de fond ont été développés : l'un pour prendre en charge l'égalité, basé sur l'E-unification rigide, et l'autre pour offrir une gestion générale des théories axiomatisées avec la déduction modulo théorie.

Nous avons mis particulièrement l'accent sur la concurrence, étant donné sa dépendance à l'égard de la théorie qu'elle aborde, en soulignant les aspects propices à la parallélisation et leurs interactions critiques avec le processus de recherche de preuve. Par exemple, le moteur de raisonnement sur l'égalité exploite la concurrence pour tenter de trouver une solution aussi rapidement que possible, tandis que la parallélisation dans le module de déduction modulo théorie est principalement appliquée à la recherche de règles de réécriture applicables.

Certification de preuves

Enfin, afin de renforcer l'intégration avec les assistants de preuve, nous avons exploré des méthodes pour générer des preuves vérifiables, y compris lors de l'utilisation de règles de Skolémisation plus avancées. Le mécanisme de déskolémisation, en assurant l'équivalence entre des preuves de formats différents, permet une recherche efficace de preuves utilisant des techniques de Skolémisation optimisées tout en préservant l'un des avantages fondamentaux des méthodes de tableau : la production d'une preuve. Cette traduction transforme le prouveur automatique de théorèmes en un générateur de certificats de preuve, lesdits certificats étant ensuite soumis à la validation d'un outil externe.

Cette approche permet non seulement d'améliorer les performances de Goéland, mais aussi de maintenir la compatibilité avec les logiciels qui utilisent des formules non modifiées. Ainsi, deux extensions ont été développées pour exporter ces preuves aux formats Coq et Lambdapi, afin qu'elles soient ensuite vérifiées par leurs outils respectifs.

Perspectives

Bien que les résultats pratiques ne puissent pas encore rivaliser avec d'autres prouveurs de pointe, ils restent prometteurs, en particulier dans le cas de la déduction modulo théorie. Cependant, plusieurs domaines d'amélioration potentielle peuvent être envisagés.

Equité dans la recherche de preuves basée sur les tableaux et résultats pratiques

Les aspects théoriques de la gestion de l'équité dans la procédure sont dignes d'intérêt, car ils répondent à des défis connus dans ce domaine. Pour que Goéland soit adapté aux applications du monde réel, les efforts futurs se concentreront sur le renforcement de la base du prouveur, y compris via une comparaison avec une version à mémoire partagée. Il est également essentiel de retravailler la gestion de la mémoire, car le prouveur rencontre actuellement des difficultés avec des fichiers de problèmes volumineux. Par ailleurs, une analyse plus approfondie des cas d'échecs doit être réalisée, étant donné que Goéland n'est pas toujours en mesure de trouver une preuve pour certains problèmes qui sont généralement considérés comme « faciles ». Bien que dans certains cas, il puisse en trouver une à l'aide de stratégies de Skolémisation plus avancées ou en essayant une substitution différente plus tôt dans la recherche de preuve, il existe toujours des problèmes qui représentent un défi pour le prouveur.

En outre, l'utilisation d'heuristiques peut améliorer le processus de recherche de preuves. Actuellement, lorsqu'un problème contient un nombre important d'axiomes qui ne peuvent pas être traduits en règles de réécriture, ils sont traités dans un ordre arbitraire, sans inclinaison spécifique. Il est donc envisageable d'implémenter une heuristique qui associe un score de priorité à chaque formule à traiter, en fonction de sa proximité avec la conjecture. Un raisonnement similaire peut être appliqué au choix de la substitution. Dans l'implémentation actuelle, le prouveur sélectionne une substitution arbitraire parmi celles qui ont le moins de variables libres non-locales, mais ce processus pourrait être amélioré en prenant en compte le degré de réintroduction de chaque variable.

Ces optimisations peuvent également impliquer une réévaluation de l'approche de fermeture anticipée. Une possibilité consiste à sacrifier la complétude de l'outil en ne considérant que les réponses initiales de chaque branche. Cela peut augmenter le taux d'échec des substitutions et, par conséquent, entraîner des retours sur trace plus fréquents, mais cela offre aussi la possibilité d'obtenir une solution plus rapidement. D'autre part, une approche alternative consiste à explorer le report de la règle de cloture, conformément au concept de fermeture progressive proposé par Giese [136]. Cette stratégie vise à réduire le nombre de retours sur trace en tentant des substitutions à chaque étape de recherche de preuve tout en permettant aux branches de se poursuivre. Des tests doivent être effectués pour ces deux approches, pour lesquelles les mises en œuvre sont en cours.

Enfin, l'un des principaux défis du raisonnement automatisé est son automatisation inhérente, qui le rend relativement statique et dépourvu d'intuition humaine. L'incorporation de méthodes d'apprentissage automatique pourrait potentiellement infuser un niveau d'« intuition humaine », suggérant la bonne substitution à essayer,

fournissant une limite de réintroduction plus précise, ou guidant la sélection d'une formule utile à traiter lors de l'étape suivante de recherche de preuves. Cette pratique s'est avérée prometteuse dans certains prouveurs de théorèmes automatisés et solveurs SMT [155, 240].

Raisonnement au sein de théories en logique du premier ordre

L'efficacité du module de raisonnement sur l'égalité est une priorité pour Goéland. Cela peut impliquer l'optimisation de l'implémentation actuelle ou l'examen d'approches alternatives, telles que le travail en cours axé sur l'ajout de règles d'égalité directement intégrées dans le calcul des tableaux. De plus, nous pouvons étudier les interactions entre la déduction modulo théorie et l'égalité, en particulier dans le but d'intégrer la réécriture de termes. En effet, comme les deux mécanismes travaillent avec le même ensemble de termes, leurs interactions pourraient être complexes.

De plus, bien que Goeland puisse gérer des problèmes typés grâce à son encodage natif des types polymorphes, il ne bénéficie actuellement pas d'une procédure spécifique pour les théories typées, telles que l'arithmétique, pour laquelle le développement d'un raisonneur dédié utilisant une approche basée sur le simplexe combinée à un algorithme de séparation et évaluation est en cours. De plus, pour étendre l'utilisation de Goéland dans la vérification de programmes, l'extension de la gestion de la logique du premier ordre typée (TFF) à la logique du premier ordre typée étendue (TXF) pourrait être envisagée. La TXF prend en charge les tuples, les expressions conditionnelles (if-then-else) et les expressions let (let-defn-in).

Malgré ses résultats prometteurs, la théorie de la déduction modulo peut encore être améliorée. Des améliorations peuvent être apportées en concevant manuellement des règles de réécriture, soit indépendamment, soit en conjonction avec le calcul automatique à partir des axiomes. Par exemple, le projet BWare, qui repose sur la théorie des ensembles, propose des règles de réécriture conçues manuellement pour le raisonnement sur des problèmes industriels. Des tests supplémentaires sur ce benchmark, avec l'ajout du raisonnement arithmétique, peuvent fournir des informations précieuses.

En outre, si la déduction modulo théorie polarisée fait preuve d'une rapidité impressionnante, elle s'accompagne d'une perte de problèmes. Des heuristiques peuvent être développées afin d'atteindre un juste équilibre dans le calcul des règles de réécriture et de bénéficier de l'augmentation de la vitesse sans sacrifier les problèmes.

Certification de preuve

Puisque la possibilité de produire des preuves vérifiées est précieuse, il est souhaitable d'étendre la sortie à d'autres assistants de preuve, élargissant ainsi les capacités de vérification de Goéland et améliorant son interopérabilité.

De plus, la vraie question repose sur la place du processus de deskolémisation : doit-il être directement implémenté dans le prouveur ou intégré dans le vérificateur de preuves ? Dans le premier cas, on peut envisager la création d'un format normalisé pour la sortie des preuves de tableaux, éliminant ainsi la nécessité d'incorporer individuellement un processus de deskolémisation dans chaque prouveur automatique. Ces preuves pourraient ensuite être traitées par un outil spécialisé qui implémente l'algorithme de traduction dans un environnement certifié, simplifiant ainsi le processus de certification automatique des preuves de tableaux.

À l'inverse, une approche alternative consiste à implémenter directement des stratégies avancées de skolémisation dans le format lui-même, transférant la responsabilité au vérificateur de preuves. Dans ce scénario, la deskolémisation devient inutile, car les preuves peuvent être directement traduites dans l'assistant de preuve correspondant. Cette approche soulève également la question de la qualité du certificat. En effet, il est préférable qu'une preuve soit concise, en omettant les étapes de calcul, qui sont laissées au vérificateur de preuves [178]. Cependant, l'intégration d'un tel mécanisme dans un vérificateur de preuves peut représenter un défi et ne répond pas au besoin d'un format de sortie commun à partir duquel des règles de traduction communes vers des assistants de preuve spécifiques peuvent être conçues.

Appendices

Appendix A
Coq's GS3 Embedding.

```
(** The following presents the Coq file used to embed GS3 in Coq.
    Each lemma corresponds to a GS3 rule. They can be easily
    proven by importing the classical logic module. **)

Lemma goeland_ax : ∀ (P : Prop), P → ¬P → ⊥.
Lemma goeland_nottrue : ¬⊤ → ⊥.
Lemma goeland_and : ∀ (P Q : Prop),
    (P → Q → ⊥) → (P ∧ Q → ⊥).
Lemma goeland_or : ∀ (P Q : Prop),
    (P → ⊥) → (Q → ⊥) → ((P → Q) → ⊥).
Lemma goeland_imply : ∀ (P Q : Prop),
    (¬P → ⊥) → (Q → ⊥) → ((P → Q) → ⊥).
Lemma goeland_equiv : ∀ (P Q : Prop),
    (¬P → ¬Q → ⊥) → (P → Q → ⊥) → ((P ↔ Q) → ⊥).
Lemma goeland_notand : ∀ (P Q : Prop),
    (¬P → ⊥) → (¬Q → ⊥) → (¬(P ∨ Q) → ⊥).
Lemma goeland_notor : ∀ (P Q : Prop),
    (¬P → ¬Q → ⊥) → (¬(P ∨ Q) → ⊥).
Lemma goeland_notimply : ∀ (P Q : Prop),
    (P → ¬ Q → ⊥) → (¬(P → Q) → ⊥).
Lemma goeland_notequiv : ∀ (P Q : Prop),
    (¬P → Q → ⊥) → (P → ¬Q → ⊥) → (¬(P ↔ Q) → ⊥).
Lemma goeland_ex : ∀ (T : Type) (P : T -> Prop),
    (∀ (z : T), ((P z) → ⊥)) → (∃ (x : T), (P x)) → ⊥.
Lemma goeland_all : ∀ (T : Type) (P : T -> Prop) (t : T),
    ((P t) → ⊥) → ((∀ (x : T), (P x)) → ⊥).
Lemma goeland_notex : ∀ (T : Type) (P : T → Prop) (t : T),
    (¬(P t) → ⊥) → (¬(∃ (x : T), (P x)) → ⊥).
Lemma goeland_notall : ∀ (T : Type) (P : T → Prop),
    (∀ (z : T), (¬(P z) → ⊥)) → (¬(∀ (x : T), (P x)) → ⊥).
```

Figure A.1: Coq's GS3 embedding — lemmas.

```
(** Those lemmas are from the wrong side, which means that when
    proving, the rules are applied to the left side of the lemma.
    Indeed, during the proof search, only formulas on the
    right side are treated. As such, we define λ-terms
    which reverse the application of the rules. **)

Definition goeland_and_s       :=
    fun P Q c h ⇒ goeland_and P Q h c.
Definition goeland_or_s        :=
    fun P Q c h i ⇒ goeland_or P Q h i c.
Definition goeland_imp_s       :=
    fun P Q c h i ⇒ goeland_imp P Q h i c.
Definition goeland_equiv_s     :=
    fun P Q c h i ⇒ goeland_equiv P Q h i c.
Definition goeland_notand_s    :=
    fun P Q c h i ⇒ goeland_notand P Q h i c.
Definition goeland_notor_s     :=
    fun P Q c h ⇒ goeland_notor P Q h c.
Definition goeland_notimp_s    :=
    fun P Q c h ⇒ goeland_notimp P Q h c.
Definition goeland_notequiv_s  :=
    fun P Q c h i ⇒ goeland_equiv P Q h i c.
Definition goeland_all_s       :=
    fun T P t c h ⇒ goeland_all T P t h c.
Definition goeland_ex_s        :=
    fun T P c h ⇒ goeland_ex T P h c.
Definition goeland_notall_s    :=
    fun T P c h ⇒ goeland_notall T P h c.
Definition goeland_notex_s     :=
    fun T P t c h ⇒ goeland_notex T P t h c.
```

Figure A.2: Coq's GS3 embedding — reversed lemmas to follow tableau rules.

Appendix B

Detailed Results of Goéland, Goéland+DMT, Goéland+DMT+EQ, Zenon, Princess, E and Vampire over a Subset of FOF

B.1 Detailed Results of Goéland over a Subset of FOF

	Number of problems solved (Cumulatitve time in second — Average time in second)	Number of problems in the category
AGT	**6** (32 s — 5.3 s)	52
ALG	**2** (29 — 14.5 s)	220
COM	**2** (262 s — 131 s)	57
CSR	**20** (1 614 s — 80 s)	577
GEO	**53** (2 588 s — 48 s)	576
GRP	**1** (48 s — 48 s)	185
HWV	**2** (362 s — 181 s)	51
ITP	**8** (4 s — 0.5 s)	99
KLE	**3** (0.09 s — 0.03 s)	223
KRS	**30** (286 s — 9.5 s)	87
LCL	**28** (423 s — 15 s)	280
MGT	**18** (492 s — 27 s)	68
MSC	**3** (32 s — 10 s)	5
NLP	**3** (1.8 — 0.6 s)	25
NUM	**18** (88 s — 4.8 s)	649
PHI	**4** (221 s — 55 s)	9
PUZ	**4** (13 s — 3.3 s)	23
SET	**124** (2 315 s — 18.6 s)	464
SEU	**55** (845 s — 15 s)	895
SEV	**1** (16 s — 16 s)	7
SWB	**18** (549 s — 30 s)	134
SWC	**1** (0.2 s — 0.2 s)	422
SYN	**209** (251 s — 1.2 s)	288
Total	**613** (10 482 s — 17.1 s)	5396

Table B.1: Detailed experimental results of Goéland over a subset of first-order problems of the TPTP library.

B.2 Detailed Results of Goéland+DMT over a Subset of FOF

	Number of problems solved (Cumulatitve time in second)	Number of problems in the category
AGT	**2** (4.6 s — 2.3 s)	52
ALG	**2** (36 s — 18 s)	220
COM	**1** (1.5 s — 1.5 s)	57
CSR	**16** (293 s — 18 s)	577
GEO	**92** (1 968 s — 21 s)	576
GRP	**1** (128 s — 128 s)	185
HWV	**2** (200 s — 100 s)	51
ITP	**6** (1.2 s — 0.2)	99
KLE	**3** (0.1 s — 0.03 s)	223
KRS	**62** (383 s — 6.1 s)	87
LCL	**36** (323 s — 9 s)	280
MGT	**18** (115 s — 6.4 s)	68
MSC	**3** (76 s — 25 s)	5
NLP	**3** (2.8 s — 0.9 s)	25
NUM	**13** (200 s — 15 s)	649
PHI	**6** (106 s — 17 s)	9
PUZ	**4** (23 s — 5.8 s)	23
SET	**217** (1 294 s — 5.9 s)	464
SEU	**56** (1 358 s — 24 s)	895
SEV	**0**	7
SWB	**17** (120 s — 7 s)	134
SWC	**1** (0.2 s — 0.2 s)	422
SYN	**209** (285 s — 1.3 s)	288
Total	**770** (6 935 s — 9 s)	5396

Table B.2: Detailed experimental results of Goéland+DMT over a subset of first-order problems of the TPTP library.

B.3 Detailed Results of Goéland+DMT+EQ over a Subset of FOF

	Number of problems solved (Cumulatitve time in second)	Number of problems in the category
AGT	**2** (4.6 s — 2.3 s)	52
ALG	**6** (839 s — 139 s)	220
COM	**3** (26 s — 8.7 s)	57
CSR	**16** (147 s — 9 s)	577
GEO	**95** (3 052 s — 32 s)	576
GRP	**1** (2.5 s — 2.5 s)	185
HWV	**2** (342 s — 171 s)	51
ITP	**6** (2 s — 0.3)	99
KLE	**4** (41 s — 10 s)	223
KRS	**64** (376 s — 5.8 s)	87
LCL	**31** (255 s — 8.2 s)	280
MGT	**18** (287 s — 16 s)	68
MSC	**4** (90 s — 22 s)	5
NLP	**3** (1.9 s — 0.6 s)	25
NUM	**22** (592 s — 27 s)	649
PHI	**6** (376 s — 62 s)	9
PUZ	**6** (23 s — 5.8 s)	23
SET	**192** (1 972 s — 10 s)	464
SEU	**42** (796 s — 18 s)	895
SEV	**0**	7
SWB	**20** (390 s — 19 s)	134
SWC	**45** (316 s — 7 s)	422
SYN	**213** (119 s — 0.5 s)	288
Total	**801** (10 060 s — 12.5 s)	5396

Table B.3: Detailed experimental results of Goéland+DMT+EQ over a subset of first-order problems of the TPTP library.

B.4 Detailed Results of Zenon over a Subset of FOF

	Number of problems solved (Cumulatitve time in second — Average time in second)	Number of problems in the category
AGT	**18** (125 s — 6.9 s)	52
ALG	**76** (4 778 s — 62 s)	220
COM	**18** (167 s — 9.2 s)	57
CSR	**120** (218 s — 1.8 s)	577
GEO	**222** (193 s — 0.8 s)	576
GRP	**5** (6.5 s — 1.3 s)	185
HWV	**4** (8.8 s — 2.2 s)	51
ITP	**9** (3.3 s — 0.3 s)	99
KLE	**6** (0.2 s — 0.03 s)	223
KRS	**74** (522 s — 7 s)	87
LCL	**55** (166 s — 3 s)	280
MGT	**36** (180 s — 5 s)	68
MSC	**5** (0.18 s — 0.03 s)	5
NLP	**11** (1.8 s — 0.6 s)	25
NUM	**106** (380 s — 3.5 s)	649
PHI	**9** (149 s — 16 s)	9
PUZ	**16** (126 s — 7.9 s)	23
SET	**150** (558 s — 3.7 s)	464
SEU	**96** (260 s — 2.7 s)	895
SEV	**1** (0.07 s — 0.07 s)	7
SWB	**23** (331 s — 14 s)	134
SWC	**56** (483 s — 8.6 s)	422
SYN	**266** (80 s — 0.3 s)	288
Total	**1 382** (9 026 s — 6.5 s)	5396

Table B.4: Detailed experimental results of Zenon over a subset of first-order problems of the TPTP library.

B.5 Detailed Results of Zenon Modulo over a Subset of FOF

	Number of problems solved (Cumulatitve time in second — Average time in second)	Number of problems in the category
AGT	**14** (45 s — 3.2 s)	52
ALG	**58** (4 019 s — 69 s)	220
COM	**18** (135 s — 7.5 s)	57
CSR	**119** (736 s — 6.1 s)	577
GEO	**217** (705 s — 3.2 s)	576
GRP	**6** (80 s — 13 s)	185
HWV	**0**	51
ITP	**12** (18 s — 1.5 s)	99
KLE	**11** (75 s — 6.8 s)	223
KRS	**57** (948 s — 16 s)	87
LCL	**29** (2 s — 0.06 s)	280
MGT	**44** (94 s — 2.1 s)	68
MSC	**5** (0.24 s — 0.04 s)	5
NLP	**12** (32 s — 2.7 s)	25
NUM	**99** (612 s — 6.1 s)	649
PHI	**8** (149 s — 16 s)	9
PUZ	**14** (74 s — 5.3 s)	23
SET	**227** (770 s — 3 s)	464
SEU	**108** (1 112 s — 10 s)	895
SEV	**2** (0.04 s — 0.02 s)	7
SWB	**14** (1.4 s — 0.1 s)	134
SWC	**53** (490 s — 9.2 s)	422
SYN	**262** (114 s — 0.4 s)	288
Total	**1 389** (10 028 s — 7.2 s)	5396

Table B.5: Detailed experimental results of Zenon Modulo over a subset of first-order problems of the TPTP library.

B.6 Detailed Results of Princess over a Subset of FOF

	Number of problems solved (Cumulatitve time in second — Average time in second)	Number of problems in the category
AGT	**9** (87 s — 9 s)	52
ALG	**124** (531 s — 4.2 s)	220
COM	**24** (42 s — 1.7 s)	57
CSR	**112** (16 541 — 147 s)	577
GEO	**130** (1880 s — 14 s)	576
GRP	**3** (4.2 s — 1.4 s)	185
HWV	**0**	51
ITP	**12** (108 s — 9 s)	99
KLE	**13** (14 — 1.1 s)	223
KRS	**56** (151 s — 2.7 s)	87
LCL	**35** (71 s —2 s)	280
MGT	**36** (385 s — 10 s)	68
MSC	**4** (5.9 s — 1.4 s)	5
NLP	**14** (35 s — 2.5 s)	25
NUM	**202** (679 s — 3.3 s)	649
PHI	**3** (2.4 s — 0.8 s)	9
PUZ	**11** (43 s — 3.9 s)	23
SET	**220** (925 s — 4.2 s)	464
SEU	**184** (533 s — 2.9 s)	895
SEV	**1** (0.9 s — 0.9 s)	7
SWB	**44** (121 s — 2.7 s)	134
SWC	**186** (652 s — 3.5 s)	422
SYN	**198** (337 — 1.9 s)	288
Total	**1 621** (23 200 s — 14.3 s)	5396

Table B.6: Detailed experimental results of Princess over a subset of first-order problems of the TPTP library.

B.7 Detailed Results of Vampire over a Subset of FOF

	Number of problems solved (Cumulatitve time in second — Average time in second)	Number of problems in the category
AGT	**52** (3 152 s — 60 s)	52
ALG	**162** (980 s — 6s)	220
COM	**48** (980 s — 20 s)	57
CSR	**303** (3 108 s — 10 s)	577
GEO	**423** (2 685 s — 6.3 s)	576
GRP	**50** (1 035 s — 20 s)	185
HWV	**28** (1 450 s — 51 s)	51
ITP	**31** (1 523 s — 49 s)	99
KLE	**121** (2 806 s — 23 s)	223
KRS	**86** (1 478 s — 17 s)	87
LCL	**148** (2 041 s — 13 s)	280
MGT	**67** (24 s — 0.3 s)	68
MSC	**5** (0.15 — 0.03 s)	5
NLP	**24** (156 s — 6.5 s)	25
NUM	**394** (7 253 s — 18.4 s)	649
PHI	**9** (0.3 s — 0.03)	9
PUZ	**20** (261 s — 13 s)	23
SET	**322** (4 207 s — 13 s)	464
SEU	**424** (5 147 s — 12 s)	895
SEV	**2** (0.4 s — 0.2 s)	7
SWB	**52** (549 s — 10 s)	134
SWC	**286** (3 726 s — 13 s)	422
SYN	**285** (302 s — 1 s)	288
Total	**3 342** (42 874 s — 12.8 s)	5396

Table B.7: Detailed experimental results of Vampire over a subset of first-order problems of the TPTP library.

B.8 Detailed Results of E over a Subset of FOF

	Number of problems solved (Cumulatitve time in second — Average time in second)	Number of problems in the category
AGT	**46** (2 389 s — 51 s)	52
ALG	**162** (682 s — 4.2 s)	220
COM	**48** (691 s — 14 s)	57
CSR	**512** (6 385 s — 12 s)	577
GEO	**454** (3 527 s — 7 s)	576
GRP	**88** (1 376 s — 15 s)	185
HWV	**12** (1 364 s — 113 s)	51
ITP	**34** (1 124 s — 33 s)	99
KLE	**179** (2 235 s — 12 s)	223
KRS	**85** (861 s — 9.9 s)	87
LCL	**182** (3 995 s — 21 s)	280
MGT	**67** (21 s — 0.3 s)	68
MSC	**5** (0.1 s — 0.02 s)	5
NLP	**25** (40 s — 1.5 s)	25
NUM	**440** (6 522 s — 14.8 s)	649
PHI	**9** (0.15 s — 0.01 s)	9
PUZ	**21** (20 s — 0.9 s)	23
SET	**362** (2 192 s — 6.1 s)	464
SEU	**489** (4 283 s — 8.7 s)	895
SEV	**3** (4.2 s — 1.4 s)	7
SWB	**80** (562 s — 7 s)	134
SWC	**353** (1 756 s — 4.9 s)	422
SYN	**283** (201 s - 0.7 s)	288
Total	**3 939** (39 638 s — 10.1 s)	5396

Table B.8: Detailed experimental results of E over a subset of first-order problems of the TPTP library.

References

[1] M. Aigner, A. Biere, C. M. Kirsch, A. Niemetz, and M. Preiner. “Analysis of Portfolio-Style Parallel SAT Solving on Current Multi-Core Architectures.” In: *POS@ SAT* 29 (2013), pp. 28–40 (cit. on p. 26).

[2] S. Anantharaman and N. Andrianarivelo. “Heuristical criteria in refutational theorem proving”. In: *Design and Implementation of Symbolic Computation Systems: International Symposium DISCO'90 Capri, Italy, April 10–12, 1990 Proceedings*. Springer. 1990, pp. 184–193 (cit. on p. 31).

[3] P. B. Andrews. “Theorem proving via general matings”. In: *Journal of the ACM (JACM)* 28.2 (1981), pp. 193–214 (cit. on p. 29).

[4] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. “Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory”. In: *Manuscript http://www. lsv. fr/~ dowek/Publi/expressing. pdf* (2016) (cit. on p. 118).

[5] O. L. Astrachan. “METEOR: Exploring model elimination theorem proving”. In: *Journal of Automated Reasoning* 13 (1994), pp. 283–296 (cit. on pp. 29, 30).

[6] O. L. Astrachan and M. E. Stickel. “Caching and lemmaizing in model elimination theorem provers”. In: *International Conference on Automated Deduction*. Springer. 1992, pp. 224–238 (cit. on p. 24).

[7] J. Avigad. “Eliminating Definitions and Skolem Functions in First-Order Logic”. In: *16th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2001, pp. 139–146 (cit. on p. 117).

[8] M. Baaz and C. G. Fermüller. “Non-elementary Speedups between Different Versions of Tableaux”. In: *TABLEAUX '95*. Ed. by P. Baumgartner, R. Hähnle, and J. Posegga. Vol. 918. Lecture Notes in Computer Science. Springer, 1995, pp. 217–230 (cit. on pp. 15, 24, 120).

[9] M. Baaz, S. Hetzl, and D. Weller. “On the Complexity of Proof Deskolemization”. In: *J. Symb. Log.* 77.2 (2012), pp. 669–686 (cit. on p. 117).

[10] L. Bachmair, N. Dershowitz, and D. A. Plaisted. “Completion without failure”. In: *Rewriting Techniques*. Elsevier, 1989, pp. 1–30 (cit. on p. 28).

[11] L. Bachmair and H. Ganzinger. “Rewrite-based equational theorem proving with selection and simplification”. In: *Journal of Logic and Computation* 4.3 (1994), pp. 217–247 (cit. on p. 11).

[12] L. Bachmair, H. Ganzinger, and A. Voronkov. “Elimination of equality via transformation with ordering constraints”. In: *International Conference on Automated Deduction*. Springer. 1998, pp. 175–190 (cit. on p. 76).

[13] P. Backeman and P. Rümmer. “Efficient algorithms for bounded rigid E-unification”. In: *Automated Reasoning with Analytic Tableaux and Related Methods: 24th International Conference, TABLEAUX 2015, Wroclaw, Poland, September 21-24, 2015, Proceedings 24*. Springer. 2015, pp. 70–85 (cit. on p. 34).

[14] P. Backeman and P. Rümmer. “Free variables and theories: Revisiting rigid e-unification”. In: *Frontiers of Combining Systems: 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015, Proceedings 10*. Springer. 2015, pp. 3–13 (cit. on p. 34).

[15] P. Backeman and P. Rümmer. “Theorem Proving with Bounded Rigid E-Unification”. In: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. Ed. by A. P. Felty and A. Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, 2015, pp. 572–587 (cit. on pp. 34, 76).

[16] T. Balyo and C. Sinz. “Parallel satisfiability”. In: *Handbook of Parallel Constraint Reasoning* (2018), pp. 3–29 (cit. on p. 26).

[17] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. “cvc5: A Versatile and Industrial-Strength SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by D. Fisman and G. Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442 (cit. on p. 34).

[18] H. P. Barendregt and E. Barendsen. “Autarkic computations in formal proofs”. In: *Journal of Automated Reasoning* 28 (2002), pp. 321–336 (cit. on pp. 89, 133).

[19] H. P. Barendregt, W. Dekkers, and R. Statman. *Lambda calculus with types*. Cambridge University Press, 2013 (cit. on p. 132).

[20] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al. “The Coq proof assistant reference manual: Version 6.1”. PhD thesis. Inria, 1997 (cit. on pp. 118, 132).

[21] C. W. Barrett. “" Decision Procedures: An Algorithmic Point of View," by Daniel Kroening and Ofer Strichman, Springer-Verlag, 2008.” In: *J. Autom. Reason.* 51.4 (2013), pp. 453–456 (cit. on p. 35).

[22] C. W. Barrett, L. De Moura, and A. Stump. “SMT-COMP: Satisfiability modulo theories competition”. In: *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17*. Springer. 2005, pp. 20–23 (cit. on p. 34).

[23] P. Baumgartner. “A model elimination calculus with built-in theories”. In: *GWAI-92: Advances in Artificial Intelligence: 16th German Conference on Artificial Intelligence Bonn, Germany, August 31–September 3, 1992 Proceedings*. Springer. 1992, pp. 30–42 (cit. on p. 32).

[24] P. Baumgartner. “Linear and unit-resulting refutations for Horn theories”. In: *Journal of Automated Reasoning* 16 (1996), pp. 241–319 (cit. on p. 36).

[25] P. Baumgartner, N. Eisinger, and U. Furbach. “A confluent connection calculus”. In: *Conference on Automated Deduction (CADE)*. Vol. 1632. Lecture Notes in Computer Science (LNCS). Springer. 1999, pp. 329–343 (cit. on p. 25).

[26] P. Baumgartner, A. Fuchs, and C. Tinelli. “(LIA)-model evolution with linear integer arithmetic constraints”. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2008, pp. 258–273 (cit. on pp. 25, 35).

[27] P. Baumgartner, U. Furbach, and I. Niemelä. “Hyper tableaux”. In: *European Workshop on Logics in Artificial Intelligence*. Springer. 1996, pp. 1–17 (cit. on pp. 28, 29).

[28] P. Baumgartner, U. Furbach, and U. Petermann. *A unified approach to theory reasoning*. Universität Koblenz-Landau. Institut für Informatik, 1992 (cit. on pp. 32, 76).

[29] P. Baumgartner and U. Petermann. “Theory reasoning”. In: *Automated Deduction. A Basis for Applications* 1 (1998), pp. 191–224 (cit. on pp. 32, 76).

[30] B. Beckert. “Depth-first proof search without backtracking for free-variable clausal tableaux”. In: *Journal of Symbolic Computation* 36.1-2 (2003), pp. 117–138 (cit. on p. 25).

[31] B. Beckert. “Semantic Tableaux with Equality”. In: *J. Log. Comput.* 7.1 (1997), pp. 39–58 (cit. on p. 83).

[32] B. Beckert. “Using E-Unification to Handle Equality in Universal Formula Semantic Tableaux”. In: *Proceedings, Theory Reasoning in Automated Deduction, Workshop at CADE-12, Nancy, France*. 1994 (cit. on pp. 33, 79, 83).

[33] B. Beckert and R. Hähnle. “An improved method for adding equality to free variable semantic tableaux”. In: *International Conference on Automated Deduction*. Springer. 1992, pp. 507–521 (cit. on pp. 33, 79, 98).

[34] B. Beckert and R. Hähnle. “Analytic tableaux”. In: *Automated Deduction: A Basis for Applications* 1 (1998), pp. 11–41 (cit. on pp. 38, 39, 44).

[35] B. Beckert, R. Hähnle, P. Oel, and M. Sulzmann. “The tableau-based theorem prover 3 TAP Version 4.0”. In: *Automated DeductionCade-13: 13th International Conference on Automated Deduction New Brunswick, NJ, USA, July 30–August 3, 1996 Proceedings 13*. Springer. 1996, pp. 303–307 (cit. on pp. 28, 36).

[36] B. Beckert, R. Hähnle, and P. H. Schmitt. “The even more liberalized δ-rule in free variable Semantic Tableaux”. In: *Computational Logic and Proof Theory*. Ed. by G. Gottlob, A. Leitsch, and D. Mundici. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 108–119 (cit. on pp. 15, 24, 94, 120).

[37] B. Beckert and C. Pape. “Incremental theory reasoning methods for semantic tableaux”. In: *Theorem Proving with Analytic Tableaux and Related Methods: 5th International Workshop, TABLEAUX'96 Terrasini, Palermo, Italy, May 15–17, 1996 Proceedings 5*. Springer. 1996, pp. 93–109 (cit. on p. 36).

[38] B. Beckert and J. Posegga. “leanTAP: Lean tableau-based deduction”. In: *Journal of Automated Reasoning* 15.3 (1995), pp. 339–358 (cit. on pp. 25, 28, 44).

[39] C. Benzmüller, M. Kerber, M. Jamnik, and V. Sorge. “Experiments with an Agent-Oriented Reasoning System”. In: *Annual Conference on Artificial Intelligence*. Vol. 2174. Lecture Notes in Computer Science (LNCS). Springer, 2001, pp. 409–424 (cit. on p. 29).

[40] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004 (cit. on p. 118).

[41] E. W. Beth. *Formal Methods: An Introduction to Symbolic Logic and to the Study of Effective Operations in Arithmetic and Logic*. Vol. 4. Synthese Library. D. Reidel Pub. Co., 1962 (cit. on pp. 11, 13).

[42] W. Bibel. “A comparative study of several proof procedures”. In: *Artificial Intelligence* 18.3 (1982), pp. 269–293 (cit. on p. 29).

[43] W. Bibel. *Automated theorem proving*. Vieweg, 1982 (cit. on pp. 34, 79, 117, 121).

[44] J.-P. Billon. “The disconnection method: a confluent integration of unification in the analytic framework”. In: *Tableaux*. Vol. 6. 1996, pp. 110–126 (cit. on p. 25).

[45] N. Bjøner, G. Reger, M. Suda, and A. Voronkov. “AVATAR modulo theories”. In: *2nd Global Conference on Artificial Intelligence*. 2016, pp. 39–52 (cit. on p. 34).

[46] J. C. Blanchette, S. Böhme, and L. C. Paulson. “Extending Sledgehammer with SMT solvers”. In: *Journal of automated reasoning* 51.1 (2013), pp. 109–128 (cit. on p. 34).

[47] J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. “Encoding Monomorphic and Polymorphic Types”. In: *Log. Methods Comput. Sci.* 12.4 (2016) (cit. on pp. 37, 106).

[48] J. C. Blanchette and A. Paskevich. “TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism”. In: *CADE*. Vol. 7898. Lecture Notes in Computer Science. Springer, 2013, pp. 414–420 (cit. on p. 107).

[49] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. “Why3: Shepherd your herd of provers”. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. 2011, pp. 53–64 (cit. on p. 37).

[50] F. Bobot and A. Paskevich. “Expressing polymorphic types in a many-sorted language”. In: *International Symposium on Frontiers of Combining Systems*. Springer. 2011, pp. 87–102 (cit. on p. 37).

[51] S. Böhme and T. Nipkow. “Sledgehammer: Judgement Day”. In: *Proceedings of the 5th International Joint Conference on Automated Reasoning*. Ed. by J. Giesl and R. Haehnle. Lecture Notes in Artificial Intelligence 6173. 2010, pp. 107–121 (cit. on p. 75).

[52] M. P. Bonacina. “A model and a first analysis of distributed-search contraction-based strategies”. In: *Annals of Mathematics and Artificial Intelligence* 27 (1999), pp. 149–199 (cit. on p. 30).

[53] M. P. Bonacina. “A Taxonomy of Parallel Strategies for Deduction”. In: *Annals of Mathematics and Artificial Intelligence* 29.1 (2000), pp. 223–257 (cit. on pp. 26, 29, 30).

[54] M. P. Bonacina. “A taxonomy of theorem-proving strategies”. In: *Artificial Intelligence Today: Recent Trends and Developments*. Springer, 1999, pp. 43–84 (cit. on p. 26).

[55] M. P. Bonacina. “Analysis of distributed-search contraction-based strategies”. In: *European Workshop on Logics in Artificial Intelligence*. Springer. 1998, pp. 107–121 (cit. on p. 30).

[56] M. P. Bonacina. “Combination of Distributed Search and Multi-Search in Peers-mcd. d: System Description”. In: *International Joint Conference on Automated Reasoning*. Springer. 2001, pp. 448–452 (cit. on pp. 26, 30, 31).

[57] M. P. Bonacina et al. “Distributed automated deduction”. PhD thesis. State University of New York at Stony Brook, 1992 (cit. on pp. 26, 30, 31).

[58] M. P. Bonacina. “On the reconstruction of proofs in distributed theorem proving: a modified Clause-Diffusion method”. In: *Journal of Symbolic Computation* 21.4 (1996), pp. 507–522 (cit. on p. 31).

[59] M. P. Bonacina. “Parallel Theorem Proving”. In: *Handbook of Parallel Constraint Reasoning*. Springer, 2018, pp. 179–235 (cit. on pp. 26, 29, 31).

[60] M. P. Bonacina, U. Furbach, and V. Sofronie-Stokkermans. “On first-order model-based reasoning”. In: *Logic, Rewriting, and Concurrency: Essays Dedicated to José Meseguer on the Occasion of His 65th Birthday* (2015), pp. 181–204 (cit. on p. 26).

[61] M. P. Bonacina and J. Hsiang. “Distributed deduction by Clause-Diffusion: distributed contraction and the Aquarius prover”. In: *Journal of Symbolic Computation* 19.1-3 (1995), pp. 245–267 (cit. on p. 31).

[62] M. P. Bonacina and J. Hsiang. “Distributed deduction by clause-diffusion: The Aquarius prover”. In: *Design and Implementation of Symbolic Computation Systems: International Symposium, DISCO’93 Gmunden, Austria, September 15–17, 1993 Proceedings*. Springer. 1993, pp. 272–287 (cit. on p. 31).

[63] M. P. Bonacina and J. Hsiang. “Parallelization of deduction strategies: an analytical study”. In: *Journal of Automated Reasoning* 13.1 (1994), pp. 1–33 (cit. on pp. 26, 30).

[64] M. P. Bonacina and W. W. McCune. “Distributed theorem proving by Peers”. In: *International Conference on Automated Deduction*. Springer. 1994, pp. 841–845 (cit. on p. 31).

[65] R. Bonichon, D. Delahaye, and D. Doligez. “Zenon: An extensible automated theorem prover producing checkable proofs”. In: *Logic for Programming, Artificial Intelligence and Reasoning*. Springer. 2007, pp. 151–165 (cit. on pp. 24, 25, 28, 33, 36, 44, 84, 118, 132, 138).

[66] R. Bonichon and O. Hermant. “A Syntactic Soundness Proof for Free-Variable Tableaux with on-the-fly Skolemization”. 2013 (cit. on pp. 118, 121).

[67] S. Bose, E. Clarke, D. Long, and S. Michaylov. “PARTHENON: a parallel theorem prover for nonHorn clauses”. In: *Fourth Annual Symposium on Logic in Computer Science*. 1989, pp. 80–89 (cit. on pp. 29, 30).

[68] D. Brand. “Proving Theorems with the Modification Method”. In: *SIAM J. Comput.* 4.4 (1975), pp. 412–430 (cit. on p. 76).

[69] P. Brauner, C. Houtmann, and C. Kirchner. “Principles of superdeduction”. In: *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. IEEE. 2007, pp. 41–50 (cit. on p. 35).

[70] R. J. Browne. “Ground term rewriting in semantic tableau systems for first order logic with equality”. PhD thesis. University of Maryland, College Park, 1987 (cit. on p. 33).

[71] A. Buch, T. Hillenbrand, and R. Fettig. “WALDMEISTER: High Performance Equational Theorem Proving”. In: *Proceedings of the 4th International Symposium on Design and Implementation of Symbolic Computation Systems*. Ed. by J. Calmet and C. Limongelli. Vol. 1128. LNCS. 1996, pp. 63–64 (cit. on p. 27).

[72] G. Burel. “Experimenting with Deduction Modulo”. In: *Automated Deduction – CADE-23*. Ed. by N. Bjørner and V. Sofronie-Stokkermans. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 162–176 (cit. on pp. 36, 84).

[73] G. Burel, G. Bury, R. Cauderlier, D. Delahaye, P. Halmagrand, and O. Hermant. “First-Order Automated Reasoning with Theories: When Deduction Modulo Theory Meets Practice”. In: *Journal of Automated Reasoning (JAR)* 64.6 (2020), pp. 1001–1050 (cit. on pp. 37, 86, 89, 107).

[74] G. Burel and C. Kirchner. “Regaining cut admissibility in deduction modulo using abstract completion”. In: *Information and Computation* 208.2 (2010), pp. 140–164 (cit. on p. 95).

[75] G. Bury, S. Cruanes, and D. Delahaye. “SMT solving modulo tableau and rewriting theories”. In: *SMT: Satisfiability Modulo Theories*. 2018 (cit. on p. 37).

[76] G. Bury, S. Cruanes, D. Delahaye, and P.-L. Euvrard. “An Automation-Friendly Set Theory for the B Method”. In: *Abstract State Machines, Alloy, B, VDM, and Z (ABZ)*. Vol. 10817. Lecture Notes in Computer Science (LNCS). Springer, 2018, pp. 409–414 (cit. on p. 138).

[77] G. Bury and D. Delahaye. “Integrating simplex with tableaux”. In: *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer. 2015, pp. 86–101 (cit. on p. 35).

[78] G. Bury, D. Delahaye, D. Doligez, P. Halmagrand, and O. Hermant. “Automated Deduction in the B Set Theory using Typed Proof Search and Deduction Modulo”. In: *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. Ed. by A. Fehnker, A. McIver, G. Sutcliffe, and A. Voronkov. Vol. 35. EPiC Series in Computing. EasyChair, 2015, pp. 42–58 (cit. on pp. 37, 86, 107, 138).

[79] R. Caferra and N. Zabel. “Building models by using tableaux extended by equational problems”. In: *Journal of Logic and Computation* 3.1 (1993), pp. 3–25 (cit. on p. 24).

[80] J. Cailler, J. Rosain, D. Delahaye, S. Robillard, and H. L. Bouziane. “Goéland: a Concurrent Tableau-Based Theorem Prover (System Description)”. In: *IJCAR 2022-11th International Joint Conference on Automated Reasoning*. Vol. 13385. 2022, pp. 359–368 (cit. on p. 100).

[81] D. Cantone and M. N. Asmundo. “A Further and Effective Liberalization of the δ-Rule in Free Variable Semantic Tableaux”. In: *Automated Deduction in Classical and Non-Classical Logics, Selected Papers*. Ed. by R. Caferra and G. Salzer. Vol. 1761. Lecture Notes in Computer Science. Springer, 1998, pp. 109–125 (cit. on pp. 15, 24, 120).

[82] D. Cantone, M. N. Asmundo, and E. G. Omodeo. “Global Skolemization with Grouped Quantifiers.” In: *APPIA-GULP-PRODE*. 1997, pp. 405–414 (cit. on p. 24).

[83] R. Cauderlier and P. Halmagrand. “Checking Zenon modulo proofs in Dedukti”. In: *arXiv preprint arXiv:1507.08719* (2015) (cit. on p. 132).

[84] K. Chaudhuri, M. Manighetti, and D. Miller. “A Proof-Theoretic Approach to Certifying Skolemization”. In: *CPP 2019*. Ed. by A. Mahboubi and M. O. Myreen. ACM, 2019, pp. 78–90 (cit. on p. 118).

[85] K. Chaudhuri and F. Pfenning. “A focusing inverse method theorem prover for first-order linear logic”. In: *International Conference on Automated Deduction*. Springer. 2005, pp. 69–83 (cit. on p. 29).

[86] A. Church. “An Unsolvable Problem of Elementary Number Theory”. In: *American Journal of Mathematics* 58.2 (Apr. 1936), pp. 345–363 (cit. on p. 7).

[87] K. Claessen, A. Lillieström, and N. Smallbone. "Sort it out with monotonicity: Translating between many-sorted and unsorted first-order logic". In: *Automated Deduction–CADE-23: 23rd International Conference on Automated Deduction, Wrocaw, Poland, July 31-August 5, 2011. Proceedings 23*. Springer. 2011, pp. 207–221 (cit. on p. 37).

[88] E. G. Coffman, M. Elphick, and A. Shoshani. "System Deadlocks". In: *ACM Comput. Surv.* 3.2 (1971), pp. 67–78 (cit. on p. 17).

[89] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout. "Alt-Ergo 2.2". In: *SMT Workshop: International Workshop on Satisfiability Modulo Theories*. 2018 (cit. on pp. 34, 37).

[90] R. Cori and D. Lascar. *Mathematical Logic: Part 1: Propositional Calculus, Boolean Algebras, Predicate Calculus, Completeness Theorems*. OUP Oxford, 2000 (cit. on p. 7).

[91] J.-F. Couchot and S. Lescuyer. "Handling polymorphism in automated deduction". In: *International Conference on Automated Deduction*. Springer. 2007, pp. 263–278 (cit. on p. 37).

[92] J. M. Crawford and L. D. Auton. "Experimental results on the crossover point in random 3-SAT". In: *Artificial intelligence* 81.1-2 (1996), pp. 31–57 (cit. on p. 28).

[93] S. Cruanes. "Superposition with structural induction". In: *International Symposium on Frontiers of Combining Systems*. Springer. 2017, pp. 172–188 (cit. on pp. 28, 37).

[94] M. D'Agostino, D. M. Gabbay, R. Hähnle, and J. Posegga. *Handbook of tableau methods*. Springer, 1999 (cit. on pp. 25, 63).

[95] M. D'Agostino and M. Mondadori. "The taming of the cut. Classical refutations with analytic cut". In: *Journal of Logic and Computation* 4.3 (1994), pp. 285–319 (cit. on p. 95).

[96] M. Davis, G. Logemann, and D. Loveland. "A Machine Program for Theorem-Proving". In: *Commun. ACM* 5.7 (1962), pp. 394–397 (cit. on p. 11).

[97] M. Davis and H. Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (1960), pp. 201–215 (cit. on pp. 11, 27).

[98] L. De Moura and N. Bjørner. "Efficient E-matching for SMT solvers". In: *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21*. Springer. 2007, pp. 183–198 (cit. on p. 35).

[99] L. De Moura and N. Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340 (cit. on p. 34).

[100] A. Degtyarev and A. Voronkov. "Equality elimination for the tableau method". In: *International Symposium on Design and Implementation of Symbolic Computation Systems*. Springer. 1996, pp. 46–60 (cit. on p. 76).

[101] A. Degtyarev and A. Voronkov. "The inverse method". In: *Handbook of Automated Reasoning*. Elsevier BV, 2001, pp. 179–272 (cit. on p. 29).

[102] A. Degtyarev and A. Voronkov. "The undecidability of simultaneous rigid E-unification". In: *Theoretical Computer Science* 166.1-2 (1996), pp. 291–300 (cit. on p. 34).

[103] A. Degtyarev and A. Voronkov. “What you always wanted to know about rigid E-unification”. In: *Journal of Automated Reasoning* 20 (1998), pp. 47–80 (cit. on pp. 34, 76, 79, 81).

[104] D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, and O. Hermant. “Zenon Modulo: When Achilles Outruns the Tortoise Using Deduction Modulo”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by K. McMillan, A. Middeldorp, and A. Voronkov. Springer Berlin Heidelberg, 2013, pp. 274–290 (cit. on pp. 36, 84, 138).

[105] D. Delahaye, C. Dubois, C. Marché, and D. Mentré. “The BWare Project: Building a Proof Platform for the Automated Verification of B Proof Obligations”. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Ed. by Y. Ait Ameur and K.-D. Schewe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 290–293 (cit. on p. 75).

[106] J. Denzinger and I. Dahn. “Cooperating theorem provers”. In: *Automated DeductionA Basis for Applications: Volume II: Systems and Implementation Techniques*. Springer, 1998, pp. 383–416 (cit. on pp. 26, 30).

[107] J. Denzinger and D. Fuchs. “Cooperation of heterogeneous provers”. In: *IJCAI*. Vol. 99. Citeseer. 1999, pp. 10–15 (cit. on p. 31).

[108] J. Denzinger and M. Fuchs. “Goal oriented equational theorem proving using team work”. In: *Annual Conference on Artificial Intelligence*. Springer. 1994, pp. 343–354 (cit. on p. 31).

[109] J. Denzinger, M. Kronenburg, and S. Schulz. “DISCOUNT – A Distributed and Learning Equational Prover”. In: *Journal of Automated Reasoning* 18.2 (1997), pp. 189–198 (cit. on p. 31).

[110] N. Dershowitz and Z. Manna. “Proving Termination with Multiset Orderings”. In: *Commun. ACM* 22 (1979), pp. 465–476 (cit. on p. 65).

[111] E. W. Dijkstra. “Solution of a problem in concurrent programming control”. In: *Pioneers and Their Contributions to Software Engineering: sd&m Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions*. Springer. 2001, pp. 289–294 (cit. on p. 19).

[112] G. Dowek. “Deduction modulo theory”. In: *arXiv preprint arXiv:1501.06523* (2015) (cit. on pp. 86, 95).

[113] G. Dowek. “What is a Theory?” In: *STACS 2002: 19th Annual Symposium on Theoretical Aspects of Computer Science Antibes-Juan les Pins, France, March 14–16, 2002 Proceedings 19*. Springer. 2002, pp. 50–64 (cit. on p. 90).

[114] G. Dowek, T. Hardin, and C. Kirchner. “HOL-$\lambda\sigma$: An Intentional First-Order Expression of Higher-Order Logic”. In: vol. 11. Nov. 1999, pp. 672–672 (cit. on pp. 36, 84).

[115] G. Dowek, T. Hardin, and C. Kirchner. “Theorem Proving Modulo”. In: *Journal of Automated Reasoning (JAR)* 31.1 (2003), pp. 33–72 (cit. on pp. 35, 76).

[116] G. Dowek and A. Miquel. “Cut elimination for Zermelo set theory”. In: 2006 (cit. on pp. 36, 84).

[117] G. Dowek and B. Werner. “Arithmetic as a Theory Modulo”. In: *Term Rewriting and Applications*. Ed. by J. Giesl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 423–437 (cit. on pp. 36, 84).

[118] B. Dutertre. “Yices 2.2”. In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 737–744 (cit. on p. 34).

[119] Z. Esen and P. Rümmer. “TRICERA: Verifying C Programs Using the Theory of Heaps”. In: *Formal Methods in Computer-aided Design (FMCAD)*. 2022, p. 380 (cit. on p. 35).

[120] M. Färber. “A Curiously Effective Backtracking Strategy for Connection Tableaux”. In: *arXiv preprint arXiv:2106.13722* (2021) (cit. on p. 23).

[121] M. Färber and C. Kaliszyk. “No Choice: Reconstruction of First-order ATP Proofs without Skolem Functions”. In: *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning*. Ed. by P. Fontaine, S. Schulz, and J. Urban. Vol. 1635. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pp. 24–31 (cit. on p. 118).

[122] M. Fisher. “An Open Approach to Concurrent Theorem Proving”. In: *Parallel Processing for Artificial Intelligence* 3 (1997), pp. 80011– (cit. on p. 29).

[123] M. Fitting. “First Order Logic and Automated Theorem Proving”. In: Springer-Verlag, 1990. Chap. 7.5 (cit. on p. 41).

[124] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, 1990 (cit. on pp. 11, 14, 24, 25, 33, 44, 63, 69, 98, 119).

[125] M. Franssen. “Implementing rigid e-unification”. In: (2008) (cit. on p. 81).

[126] G. Frege. *Grundlagen der Arithmetik: Studienausgabe MIT Dem Text der Centenarausgabe*. Breslau: Wilhelm Koebner Verlag, 1884 (cit. on p. 7).

[127] M. Fuchs and A. Wolf. “System description: cooperation in model elimination: CPTHEO”. In: *Automated DeductionCADE-15: 15th International Conference on Automated Deduction Lindau, Germany, July 5–10, 1998 Proceedings 15*. Springer. 1998, pp. 42–46 (cit. on pp. 29, 31).

[128] U. Furbach. “Theory reasoning in first order calculi”. In: *Management and Processing of Complex Data Structures: Third Workshop on Information Systems and Artificial Intelligence Hamburg, Germany, February 28–March 2, 1994 Proceedings 3*. Springer. 1994, pp. 139–156 (cit. on pp. 32, 76).

[129] J. Gallier, P. Narendran, S. Raatz, and W. Snyder. “Theorem proving using equational matings and rigid E-unification”. In: *Journal of the ACM (JACM)* 39.2 (1992), pp. 377–430 (cit. on pp. 34, 77, 79).

[130] J. H. Gallier, S. Raatz, and W. Snyder. *Theorem proving using rigid E-unification: Equational matings*. University of Pennsylvania, School of Engineering and Applied Science , 1987 (cit. on pp. 34, 79).

[131] J. H. Gallier, W. Snyder, P. Narendran, and D. A. Plaisted. “Rigid E-Unification is NP-Complete”. In: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS ’88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, 1988, pp. 218–227 (cit. on p. 34).

[132] Y. Ge, C. Barrett, and C. Tinelli. “Solving quantified verification conditions using satisfiability modulo theories”. In: *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21*. Springer. 2007, pp. 167–182 (cit. on p. 35).

[133] Y. Ge and L. De Moura. “Complete instantiation for quantified formulas in satisfiabiliby modulo theories”. In: *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26-July 2, 2009. Proceedings 21*. Springer. 2009, pp. 306–320 (cit. on p. 35).

[134] G. Gentzen. “Untersuchungen über das logische schlieSSen. I.” In: *Mathematische zeitschrift* 35 (1935) (cit. on pp. 11, 29).

[135] M. Giese. “A model generation style completeness proof for constraint tableaux with superposition”. In: *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer. 2002, pp. 130–144 (cit. on pp. 34, 76).

[136] M. Giese. “Incremental closure of free variable tableaux”. In: *International Joint Conference on Automated Reasoning*. Springer. 2001, pp. 545–560 (cit. on pp. 25, 41, 44, 147, 158).

[137] M. Giese and W. Ahrendt. “Hilbert’s -Terms in Automated Theorem Proving”. In: *Automated Reasoning with Analytic Tableaux and Related Methods*. Ed. by N. V. Murray. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 171–185 (cit. on pp. 15, 24, 120).

[138] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur*. Ph.D. thesis, Université Paris VII, 1972 (cit. on p. 106).

[139] W. D. Goldfarb. “The undecidability of the second-order unification problem”. In: *Theoretical Computer Science* 13.2 (1981), pp. 225–230 (cit. on p. 6).

[140] R. Guerraoui and P. Kuznetsov. *Algorithms for concurrent systems*. EPFL press, 2018 (cit. on p. 16).

[141] R. Hähnle. “Tableaux and Related Methods”. In: *Handbook of Automated Reasoning (Volume 1)*. Ed. by J. A. Robinson and A. Voronkov. ISBN 0-444-50813-9. Elsevier and MIT Press, 2001, pp. 100–178 (cit. on pp. 28, 39).

[142] R. Hähnle and B. Beckert. “Proof confluent tableau calculi”. In: *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer. 1999 (cit. on p. 15).

[143] R. Hähnle and S. Klingenbeck. “A-ordered tableaux”. In: *Journal of Logic and Computation* 6.6 (1996), pp. 819–833 (cit. on p. 28).

[144] R. Hähnle and P. H. Schmitt. “The Liberalized δ-Rule in Free Variable Semantic Tableaux”. In: *J. Autom. Reason.* 13.2 (1994), pp. 211–221 (cit. on pp. 15, 24, 120).

[145] Y. Hamadi and C. Wintersteiger. “Seven challenges in parallel SAT solving”. In: *AI Magazine* 34.2 (2013), pp. 99–99 (cit. on p. 26).

[146] P. B. Hansen. *The origin of concurrent programming: from semaphores to remote procedure calls*. Springer Science & Business Media, 2013 (cit. on p. 17).

[147] J. Herbrand. “Recherches sur la théorie de la démonstration”. In: (1930) (cit. on p. 93).

[148] J. Hintikka. “Two Papers on Symbolic Logic: Form and Content in Quantification Theory and Reductions in the Theory of Types”. In: *Societas Philosophica, Acta philosophica Fennica* 8 (1955), pp. 7–55 (cit. on pp. 11, 13).

[149] C. A. R. Hoare. “Communicating Sequential Processes”. In: *Communications of the ACM* 21.8 (1978), pp. 666–677 (cit. on pp. 19, 20).

[150] H. Hojjat and P. Rümmer. “The ELDARICA horn solver”. In: *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE. 2018, pp. 1–7 (cit. on p. 35).

[151] J. Hsiang and M. Rusinowitch. “On word problems in equational theories”. In: *Automata, Languages and Programming: 14th International Colloquium Karlsruhe, Federal Republic of Germany, July 13–17, 1987 Proceedings 14*. Springer. 1987, pp. 54–71 (cit. on p. 28).

[152] G. Huet and D. C. Oppen. “Equations and rewrite rules: A survey”. In: *Formal Language Theory* (1980), pp. 349–405 (cit. on p. 65).

[153] M. Jacquel, K. Berkani, D. Delahaye, and C. Dubois. “Tableaux Modulo Theories using Superdeduction: An Application to the Verification of B Proof Rules with the Zenon Automated Theorem Prover”. In: vol. 7364. June 2012, pp. 332–338 (cit. on pp. 36, 84).

[154] R. C. Jeffrey. *Formal Logic: Its Scope and Limits*. McGraw-Hill, 1967 (cit. on p. 32).

[155] C. Kaliszyk, J. Urban, H. Michalewski, and M. Olák. “Reinforcement learning of theorem proving”. In: *Advances in Neural Information Processing Systems* 31 (2018) (cit. on pp. 148, 159).

[156] S. Kanger. “A simplified proof method for elementary logic”. In: *Studies in Logic and the Foundations of Mathematics*. Vol. 35. Elsevier, 1963, pp. 87–94 (cit. on p. 32).

[157] G. Katsirelos, A. Sabharwal, H. Samulowitz, and L. Simon. “Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 27. 1. 2013, pp. 481–488 (cit. on p. 26).

[158] J. Kay and P. Lauder. “A Fair Share Scheduler”. In: *Commun. ACM* 31.1 (1988), pp. 44–55 (cit. on p. 19).

[159] D. E. Knuth and P. B. Bendix. “Simple word problems in universal algebras”. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970* (1983), pp. 342–376 (cit. on p. 28).

[160] E. de Kogel. “Rigid E-unification simplified”. In: *International Workshop on Theorem Proving with Analytic Tableaux and Related Methods*. Springer. 1995, pp. 17–30 (cit. on p. 34).

[161] K. Konrad. “HOT: A Concurrent Automated Theorem Prover Based on Higher-Order Tableaux”. In: *Theorem Proving in Higher Order Logics (TPHOLs)*. Vol. 1479. Lecture Notes in Computer Science (LNCS). Springer, 1998, pp. 245–261 (cit. on p. 31).

[162] R. E. Korf. “Depth-First Iterative-Deepening: An Optimal Admissible Tree Search”. In: *Artificial Intelligence* 27.1 (1985), pp. 97–109 (cit. on p. 44).

[163] L. Kovács and A. Voronkov. “First-order theorem proving and Vampire”. In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 1–35 (cit. on pp. 27, 138).

[164] R. Kowalski and D. Kuehner. “Linear resolution with selection function”. In: *Artificial Intelligence* 2.3-4 (1971), pp. 227–260 (cit. on p. 27).

[165] K. R. M. Leino and P. Rümmer. “A polymorphic intermediate verification language: Design and logical encoding”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2010, pp. 312–327 (cit. on p. 37).

[166] R. Letz. “First-Order Tableau Methods”. In: *Handbook of Tableau Methods*. Ed. by M. D’Agostino, D. M. Gabbay, R. Hähnle, and J. Posegga. ISBN 978-94-017-1754-0. Springer, 1999, pp. 125–196 (cit. on pp. 11, 28).

[167] R. Letz. “Using matings for pruning connection tableaux”. In: *International Conference on Automated Deduction*. Springer. 1998, pp. 381–396 (cit. on p. 24).

[168] R. Letz, K. Mayr, and C. Goller. “Controlled integration of the cut rule into connection tableau calculi”. In: *Journal of Automated Reasoning* 13.3 (1994), pp. 297–337 (cit. on p. 24).

[169] D. W. Loveland. “Automated Theorem Proving. A Logical Basis”. In: *Journal of Symbolic Logic* 45.3 (1980) (cit. on pp. 28, 29).

[170] E. L. Lusk, W. W. McCune, and J. Slaney. “Roo: A parallel theorem prover”. In: *Automated DeductionCADE-11: 11th International Conference on Automated Deduction Saratoga Springs, NY, USA, June 15–18, 1992 Proceedings 11*. Springer. 1992, pp. 731–734 (cit. on p. 30).

[171] N. Manthey. “Towards next generation sequential and parallel SAT solvers”. In: *KI-Künstliche Intelligenz* 30.3-4 (2016), pp. 339–342 (cit. on p. 26).

[172] R. Martins, V. Manquinho, and I. Lynce. “An overview of parallel SAT solving”. In: *Constraints* 17 (2012), pp. 304–347 (cit. on p. 26).

[173] P. J. Martn, A. Gavilanes, and J. Leach. “Tableau Methods for a Logic with Term Declarations”. In: *J. Symb. Comput.* 29.2 (2000), pp. 343–372 (cit. on p. 115).

[174] S. Y. Maslov. “An inverse method for establishing deducibility of nonprenex formulas of the predicate calculus”. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Springer, 1983, pp. 48–54 (cit. on p. 29).

[175] S. Y. Maslov. *Theory of deductive systems and its applications*. MIT Press, 1987 (cit. on p. 29).

[176] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012 (cit. on p. 18).

[177] W. W. McCune. *Otter 3.0 reference manual and guide*. Tech. rep. Argonne National Lab.(ANL), Argonne, IL (United States), 1994 (cit. on pp. 27, 30, 31).

[178] D. Miller. “A proposal for broad spectrum proof certificates”. In: *Certified Programs and Proofs: First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings 1*. Springer. 2011, pp. 54–69 (cit. on pp. 133, 149, 160).

[179] D. A. Miller. “A Compact Representation of Proofs”. In: *Stud Logica* 46.4 (1987), pp. 347–370 (cit. on p. 118).

[180] G. Mints. “Resolution calculus for the first order linear logic”. In: *Journal of Logic, Language and Information* 2 (1993), pp. 59–83 (cit. on p. 29).

[181] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. “SETHEO and e-SETHEO-the CADE-13 systems”. In: *Journal of Automated Reasoning* 18 (1997), pp. 237–246 (cit. on pp. 24, 29, 31).

[182] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. “The Lean Theorem Prover (System Description)”. In: *CADE-25*. Ed. by A. P. Felty and A. Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, 2015, pp. 378–388 (cit. on p. 118).

[183] N. V. Murray and E. Rosenthal. “Inference with path resolution and semantic graphs”. In: *Journal of the ACM (JACM)* 34.2 (1987), pp. 225–254 (cit. on p. 32).

[184] N. V. Murray and E. Rosenthal. “Theory links: Applications to automated theorem proving”. In: *Journal of Symbolic Computation* 4.2 (1987), pp. 173–190 (cit. on p. 32).

[185] R. Nieuwenhuis, T. Hillenbrand, A. Riazanov, and A. Voronkov. “On the evaluation of indexing techniques for theorem proving”. In: *1st International Joint Conference on Automated Reasoning, IJCAR 2001*. Vol. 2083. Lecture Notes in Computer Science. Springer Nature, 2001, pp. 257–271 (cit. on p. 101).

[186] R. Nieuwenhuis and A. Rubio. “Paramodulation-Based Theorem Proving.” In: *Handbook of automated reasoning* 1.7 (2001), pp. 371–443 (cit. on pp. 11, 28).

[187] R. Nieuwenhuis and A. Rubio. “Theorem proving with ordering and equality constrained clauses”. In: *Journal of Symbolic Computation* 19.4 (1995), pp. 321–351 (cit. on pp. 28, 81).

[188] F. Oppacher and E. Suen. “HARP: A tableau-based theorem prover”. In: *Journal of Automated Reasoning* 4 (1988), pp. 69–100 (cit. on pp. 24, 28).

[189] J. Otten. “A non-clausal connection calculus”. In: *Automated Reasoning with Analytic Tableaux and Related Methods: 20th International Conference, TABLEAUX 2011, Bern, Switzerland, July 4-8, 2011. Proceedings 20*. Springer. 2011, pp. 226–241 (cit. on p. 29).

[190] J. Otten. “nanoCoP: A non-clausal connection prover”. In: *Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27–July 2, 2016, Proceedings 8*. Springer. 2016, pp. 302–312 (cit. on p. 29).

[191] J. Otten and W. Bibel. “leanCoP: lean connection-based theorem proving”. In: *Journal of Symbolic Computation* 36.1-2 (2003), pp. 139–161 (cit. on p. 29).

[192] R. A. Overbeek. “An implementation of hyper-resolution”. In: *Computers & Mathematics with Applications* 1.2 (1975), pp. 201–214 (cit. on p. 27).

[193] L. C. Paulson. “Natural Deduction as Higher-Order Resolution”. In: *J. Log. Program.* 3.3 (1986), pp. 237–258 (cit. on p. 118).

[194] N. Peltier. “Simplifying and generalizing formulae in tableaux. Pruning the search space and building models”. In: *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer. 1997, pp. 313–327 (cit. on p. 24).

[195] B. Pelzer and I. Glöckner. “Combining theorem proving with natural language processing”. In: *Proc. of the First Int. Workshop on Practical Aspects of Automated Reasoning (PAAR 2008), CEUR Workshop Proceedings*. 2008, pp. 71–80 (cit. on p. 36).

[196] U. Petermann. “How to build in an open theory into connection calculi”. In: *Computers and artificial intelligence* 11.2 (1992), pp. 105–142 (cit. on p. 32).

[197] G. E. Peterson. “A technique for establishing completeness results in theorem proving with equality”. In: *SIAM Journal on Computing* 12.1 (1983), pp. 82–100 (cit. on p. 28).

[198] A. Platzer. “Differential dynamic logic for hybrid systems”. In: *Journal of Automated Reasoning* 41 (2008), pp. 143–189 (cit. on p. 35).

[199] G. Plotkin. “Building-in Equational Theories”. In: *Machine Intelligence* 7 (1972), pp. 73–90 (cit. on pp. 35, 84).

[200] A. Policriti and J. T. Schwartz. “T-theorem proving I”. In: *Journal of Symbolic Computation* 20.3 (1995), pp. 315–342 (cit. on p. 32).

[201] J. Posegga. “Compiling proof search in semantic tableaux”. In: *Methodologies for Intelligent Systems: 7th International Symposium, ISMIS'93 Trondheim, Norway, June 15–18, 1993 Proceedings 7*. Springer. 1993, pp. 39–48 (cit. on p. 25).

[202] D. Prawitz. “Natural deduction: a proof-theoretical study”. PhD thesis. Almqvist & Wiksell, 1965 (cit. on pp. 35, 84).

[203] V. Prevosto and U. Waldmann. “Spass+ t”. In: *ESCoR: Empirically Successful Computerized Reasoning* 192 (2006), p. 88 (cit. on p. 34).

[204] W. Pugh. “The Omega test: a fast and practical integer programming algorithm for dependence analysis”. In: *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. 1991, pp. 4–13 (cit. on p. 35).

[205] S. V. Reeves. “Adding equality to semantic tableaux”. In: *Journal of Automated Reasoning* 3 (1987), pp. 225–246 (cit. on p. 33).

[206] G. Reger, M. Suda, and A. Voronkov. “Unification with abstraction and theory instantiation in saturation-based reasoning”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2018, pp. 3–22 (cit. on p. 34).

[207] J. C. Reynolds. “Towards a theory of type structure”. In: *Programming Symposium: Proceedings, Colloque sur la Programmation Paris, April 9–11, 1974*. Springer. 1974, pp. 408–425 (cit. on p. 106).

[208] A. J. Robinson and A. Voronkov. *Handbook of automated reasoning*. Vol. 1. Elsevier, 2001 (cit. on pp. 3, 26, 153).

[209] J. A. Robinson. “A Machine-Oriented Logic Based on the Resolution Principle”. In: *J. ACM* 12.1 (1965), pp. 23–41 (cit. on pp. 11, 27).

[210] J. A. Robinson. “Automatic Deduction with Hyper-Resolution”. In: *Journal of Symbolic Logic* 39.1 (1974), pp. 189–190 (cit. on p. 27).

[211] P. Rümmer. “A constraint sequent calculus for first-order logic with linear integer arithmetic”. In: *Logic for Programming, Artificial Intelligence and Reasoning*. Springer. 2008, pp. 274–289 (cit. on pp. 25, 35, 44, 75, 138).

[212] A. Schlichtkrull, J. C. Blanchette, and D. Traytel. “A verified prover based on ordered resolution”. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. Ed. by A. Mahboubi and M. O. Myreen. ACM, 2019, pp. 152–165 (cit. on pp. 4, 154).

[213] S. Schulz. “E–a brainiac theorem prover”. In: *Ai Communications* 15.2-3 (2002), pp. 111–126 (cit. on pp. 28, 138).

[214] J. Schumann. “Parallel Theorem Provers – An Overview”. In: *Parallelization in Inference Systems*. Vol. 590. Lecture Notes in Computer Science (LNCS). Dagstuhl Castle (Germany): Springer, 1992, pp. 26–50 (cit. on pp. 26, 30).

[215] J. Schumann. “Tableau-based theorem provers: Systems and implementations”. In: *Journal of Automated Reasoning* 13 (1994), pp. 409–421 (cit. on p. 28).

[216] J. Schumann and R. Letz. “PARTHEO: A High-Performance Parallel Theorem Prover”. In: *Conference on Automated Deduction (CADE)*. Vol. 449. Lecture Notes in Computer Science (LNCS). Springer, 1990, pp. 40–56 (cit. on pp. 29, 30).

[217] J. M. P. Schumann. “DELTAA bottom-up preprocessor for top-down theorem provers: System abstract”. In: *International Conference on Automated Deduction*. Springer. 1994, pp. 774–777 (cit. on p. 31).

[218] T. Skolem. “Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Bewiesbarkeit mathematischer Sätze nebst einem Theorem über dichte Mengen”. In: (1920) (cit. on pp. 15, 93).

[219] J. R. Slagle. “Automated theorem-proving for theories with simplifiers commutativity, and associativity”. In: *Journal of the ACM (JACM)* 21.4 (1974), pp. 622–642 (cit. on p. 28).

[220] R. M. Smullyan. “Analytic cut”. In: *The Journal of Symbolic Logic* 33.4 (1969), pp. 560–564 (cit. on p. 94).

[221] R. M. Smullyan. *First-Order Logic*. Berlin Heidelberg New York: Springer-Verlag, 1968 (cit. on p. 24).

[222] E. Speckenmeyer, B. Monien, and O. Vornberger. “Superlinear speedup for parallel backtracking”. In: *International Conference on Supercomputing*. Springer. 1987, pp. 985–993 (cit. on p. 26).

[223] A. Steen, M. Wisniewski, and C. Benzmüller. “Going polymorphic-TH1 reasoning for Leo-III”. In: *IWIL Workshop and LPAR Short Presentations*. EasyChair. 2017, p. 13 (cit. on p. 37).

[224] M. E. Stickel. “Automated deduction by theory resolution”. In: *Journal of Automated Reasoning* 1.4 (1985), pp. 333–355 (cit. on p. 32).

[225] G. Sutcliffe. “The CADE ATP System Competition - CASC”. In: *AI Magazine* 37.2 (2016), pp. 99–101 (cit. on p. 11).

[226] G. Sutcliffe. “The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0”. In: *Journal of Automated Reasoning (JAR)* 59.4 (2017), pp. 483–502 (cit. on p. 135).

[227] G. Sutcliffe and J. Pinakis. “A heterogeneous parallel deduction system”. In: *Proceedings of the Workshop on Automated Deduction: Logic Programming and Parallel Computing Approaches, FGCS*. Vol. 92. Citeseer. 1992 (cit. on p. 31).

[228] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. “The TPTP Typed First-Order Form with Arithmetic”. In: *LPAR*. 2012 (cit. on pp. 107, 108).

[229] C. B. Suttner and J. Schumann. “Parallel automated theorem proving”. In: *Machine Intelligence and Pattern Recognition*. Vol. 14. Elsevier, 1994, pp. 209–257 (cit. on pp. 26, 30).

[230] A. S. Troelstra and H. Schwichtenberg. *Basic proof theory, Second Edition*. Vol. 43. Cambridge tracts in theoretical computer science. Cambridge University Press, 2000 (cit. on p. 117).

[231] M. Tsoukalos. “Mastering Go: Create Golang production applications using network libraries, concurrency, machine learning, and advanced data structures”. In: Packt Publishing Ltd., 2019, pp. 439–463 (cit. on p. 100).

[232] A. Turing. “On computable numbers, with an application to the Entscheidungsproblem. A correction”. In: *Proceedings of the London* (1938) (cit. on p. 7).

[233] A. Voronkov. “The anatomy of vampire - Implementing bottom-up procedures with code trees”. English. In: *Journal of Automated Reasoning* 15.2 (June 1995), pp. 237–265 (cit. on p. 101).

[234] A. Voronkov. “Theorem proving in non-standard logics based on the inverse method”. In: *Automated DeductionCADE-11: 11th International Conference on Automated Deduction Saratoga Springs, NY, USA, June 15–18, 1992 Proceedings 11*. Springer. 1992, pp. 648–662 (cit. on p. 29).

[235] H. Wang. “Logic of many-sorted theories”. In: *The Journal of Symbolic Logic* 17.2 (1952), pp. 105–116 (cit. on p. 106).

[236] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topi. “Spass Version 2.0”. In: *Automated DeductionCADE-18: 18th International Conference on Automated Deduction Copenhagen, Denmark, July 27–30, 2002 Proceedings 18*. Springer. 2002, pp. 275–279 (cit. on pp. 28, 31).

[237] L. Wos and G. Robinson. “Paramodulation and set of support”. In: *Symposium on Automatic Demonstration: Held at Versailles/France, December 1968*. Springer. 1968, pp. 276–310 (cit. on p. 28).

[238] L. Wos, G. A. Robinson, D. F. Carson, and L. Shalla. “The concept of demodulation in theorem proving”. In: *Journal of the ACM (JACM)* 14.4 (1967), pp. 698–709 (cit. on p. 28).

[239] C.-H. Wu. “A Multi-Agent Framework for Distributed Theorem Proving”. In: *Expert Systems with Applications* 29.3 (2005), pp. 554–565 (cit. on p. 29).

[240] H. Zhu, S. Magill, and S. Jagannathan. “A data-driven CHC solver”. In: *ACM SIGPLAN Notices* 53.4 (2018), pp. 707–721 (cit. on pp. 148, 159).