ÉCOLE NORMALE SUPÉRIEURE DE LYON

AN INTERNSHIP REPORT
ON

Skolemisation in First-Order Logic : Certification and Optimisation

Submitted in partial fullfilment of the requirements for the award of the degree of

LICENCE 3
IN
INFORMATIQUE FONDAMENTALE

by

Johann Rosain

Under the Guidance of

Julie Cailler
David Delahaye
Olivier Hermant
Simon Robillard

Contents

1 Introduction

Proofs — sequences of sentences composing an argument — have been part of the scientists standards for millennia. They are the best (and only) way to believe the affirmations stated to describe the world. Yet, ironically, they have not always been *foolproof*. False proofs are published every year, and famous minds are not exempt of mistakes, as illustrated by Legendre's tries on the parallel postulate [29] or Hilbert's attempted proof of the continuum hypothesis [26]. It thus raises the question of a proof's trustworthiness (or *soundness*), i.e., how to believe in proofs. Hilbert and Gentzen, acting as forefathers, proposed formal systems that allow syntactical reasoning over sentences, easily providing sound proofs by construction. However, the difficulty of verifying this kind of proof by human means together with Gödel's incompleteness theorem [23] and, subsequently, Church's undecidability theorem [14], showing that there exists sentences that can not be proven in such systems, thwarted these plans. As such, it has made the semantics arguments, i.e., the arguments involving the deep meaning of the manipulated objects, being preferred over the years.

A fundamental shift in the way of proving has however occurred in the 1960s due to the discovery of the *Curry-Howard isomorphism* [16, 27], which offers an automatic way of verifying the soundness of a syntactic proof. This mechanism has birthed computer programs, called *proof assistants* [6, 31, 17], which make use of the isomorphism to mechanically check the proofs entered in their language. Nevertheless, the shift has not been immediate, as only a narrow range of specialists were convinced by their usage. However, as time advanced, the features added have enabled a wider use of those programs and, for instance, *certified* (i.e., proven to satisfy a given specification) pieces of software were produced [28]. Nowadays, more and more *critical* programs (i.e., programs that, if they fail, put human lives in danger) are developed, and the need for powerful tools to prove them has never been more keenly felt.

Although the proof assistants have seen a wider range of features over the past few decades, the usage of background reasoners could provide an effective boost for such tools to allow the users to automate the tedious bits of a proof. Automated theorem provers make a fine candidate for this kind of job as such tools [39, 35] have had an extended research history, and have been growing more and more efficient [37]. Nevertheless, to make those tools the *right* candidates, the century-old question still persists — even though now aimed towards machines and not humans — are the proofs output by automated theorem provers sound?

Indeed, as humans have implemented those tools, it is natural to expect bugs to appear somewhere in the code. In this kind of system, bugs can be disastrous, making it prove non-theorems and thus compromising the kernel of proof assistants. Fortunately, there are two ways to avoid inconsistencies in automated theorem provers: fully certify the kernel of the prover using a proof assistant or produce machine-checkable proofs. The former is a time-consuming, arduous and long-term work [34], whereas the later is, in general, more easily accessible.

This internship focuses on the second line of thought. Nonetheless, over the years, proof-search algorithms have been thoroughly optimised by using multiple semantics arguments for the preservation of the proofs

soundness. It thus led the automated theorem provers to yield proofs that can not trivially be machine-checked, as the languages of the tools involved differ greatly. Thus, the main objective of the internship has been to develop and implement a method to translate a prover's output to a proof-assistant system, i.e., to make an automated tool output machine-checkable proofs. The contributions to the field have thus been many-fold: a novel algorithm that generalises gently to multiple proof optimisations have been proposed, proved sound and implemented in an automated prover that validates its in-practice use.

As such, this document is organised as follows. Section 2 presents the detailed challenges addressed during the internship and defines the notions needed throughout it. Section 3 then proceeds to exhibit the main contributions, i.e., the proposed algorithm and its soundness proof over some optimisations. Finally, Section 4 gives an overview of the implementation process needed to certify proofs and summarises the results obtained by the algorithm in practice.

2 Context and Challenges Faced

The work carried out during this internship relies heavily on first-order logic. As such, a solid basis of this logic is needed to smoothly understand the topics discussed and is offered in Subsec. 2.1. Subsec. 2.2 then gives an overview of proof systems and focuses on two in particular: the tableaux method and GS3. Subsec. 2.3 presents the need of a translation, called *deskolemisation*, between those two systems and introduces some the state-of-the-art work. Finally, Subsec. 2.4 discusses the limitations of the existing solutions, expresses the internship's objectives and highlights the contributions made to the field.

2.1 First-Order Language

First-Order Language. As defined by Cori and Lascar in [15], a first-order logic language, denoted $\mathscr{L}$, is an alphabet composed of four disjoint sets: an infinitely countable set of variables $\mathscr{V}$, a set regrouping propositional logic connectors, quantifiers, brackets, commas and the symbols $\top$ and $\bot$ (read true and false), a set of function symbols $\mathscr{S}_F$ with each symbol having its arity associated and a set of relational symbols and their arity $\mathscr{S}_P$, also called *predicates*. The sentences of this language are called *formulas* and are built upon words called *terms* and *atomic formulas*. The definitions will be given from bottom to top: starting from the words (terms) and going all the way up to build sentences (formulas).

Definition 2.1: Term

> The set of terms of first-order logic $\mathscr{T}$ over $\mathscr{L}$ is recursively defined by the family $(\mathscr{T}_n)_{n\in\mathbb{N}}$ as follows.
>
> - $\mathscr{T}_0 = \mathscr{V} \cup \mathscr{S}_F^0$ where $\mathscr{S}_F^0$ denotes the functions of arity 0 ;
> - $\mathscr{T}_n = \mathscr{T}_{n-1} \cup \{f(t_1,\dots,t_k) \mid k \in \mathbb{N} \wedge f \in \mathscr{S}_F^k \wedge t_1,\dots,t_k \in \mathscr{T}_{n-1}\}$;
> - $\mathscr{T} = \bigcup_{n\in\mathbb{N}} \mathscr{T}_n$.

In essence, $\mathscr{T}$ is the smallest subset of words over $\mathscr{L}$ which contains constants, variables and is stable by associating any n-uple (for $n \geqslant 1$) of words to be the arguments of a function symbol of $\mathscr{S}_F^n$. Another construction can yield words and composes the first layer of the sentences of this language: the *atomic formulas*.

Definition 2.2: Atomic Formula

> A word M over $\mathscr{L}$ is an *atomic formula* if and only if there exists $n \in \mathbb{N}^*$, an n-ary relational symbol R of $\mathscr{S}_P$ and n terms $t_1,\dots,t_n$ of $\mathscr{L}$ such that $M = R(t_1,\dots,t_n)$.

To express events, those words can be combined together using logical connectors such as $\wedge$ (the conjunction), $\vee$ (the disjunction), $\Rightarrow$ (the implication) or $\neg$ (the negation) and can be quantified universally ($\forall$) or existentially ($\exists$). This combination yields a sentence, also called a *formula*.

Definition 2.3: Formula

> The set of formulas of first-order logic $\mathscr{F}$ over $\mathscr{L}$ is recursively defined by the family $(\mathscr{F}_n)_{n\in\mathbb{N}}$ as follows.
>
> - $\mathscr{F}_0$ is the set of atomic formulas plus $\top$ and $\bot$;
> - $\mathscr{F}_n = \mathscr{F}_{n-1} \cup \{\neg F \mid F \in \mathscr{F}_{n-1}\} \cup \{F_1\ \alpha\ F_2 \mid F_1, F_2 \in \mathscr{F}_{n-1}\} \cup \{\beta x.F \mid F \in \mathscr{F}_{n-1}\}$ where $\alpha \in \{\wedge,\vee,\Rightarrow\}$ and $\beta \in \{\forall,\exists\}$;
> - $\mathscr{F} = \bigcup_{n\in\mathbb{N}} \mathscr{F}_n$.

As it can be remarked, variables can be quantified using an existential ($\exists$) or a universal ($\forall$) binder, but they

can also appear without being bound. The set of unbound variables of a formula P is called *free variables* of P and is denoted $FV(P)$. A formula that does not feature free variables is said *closed*. Throughout this document, bound variables will be denoted $x, x', x_1, y, \ldots$, whereas free variables will use the same letters but capitalised $X, X', X_1, Y, \ldots$. Constants will use the start of the alphabet $a, a', a_1, b, \ldots$, n-ary functions will be denoted as usually done with $f, f', f_1, g, \ldots$ and formulas will be denoted by capital letters such as P, P', P_1, Q or eventually by greek letters $\varphi, \varphi', \varphi_1, \psi$. Furthermore, $P \iff Q$ can be used as a shortcut for $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

Syntactic Manipulations. Formulas and terms containing free variables can be syntactically manipulated to replace them by other terms using a function from $\mathscr{V}$ to $\mathscr{T}$ usually denoted σ and called *substitution*.

Definition 2.4: Term's Substitution

Let σ be a substitution and t a term of $\mathscr{T}$. Applying σ upon t is denoted $t\sigma$ and defined by induction over t.

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

Definition 2.5: Formula's Substitution

Let σ be a substitution and P a formula of $\mathscr{F}$. Applying σ upon P is denoted $P\sigma$ and defined by induction over P.

- If P is an n-ary predicate $R(t_1, \ldots, t_n)$ then $P\sigma = R(t_1\sigma, \ldots, t_n\sigma)$.
- If $P = \neg Q$ then $P\sigma = \neg(Q\sigma)$.
- If P is a binary formula $F_1 \ \alpha \ F_2$ with the connector $\alpha \in \{\wedge, \vee, \Rightarrow\}$ then $P\sigma = (F_1\sigma) \ \alpha \ (F_2\sigma)$.
- If P is a formula $\beta x.Q$ quantified by $\beta \in \{\forall, \exists\}$ then there are two cases:
  - if $x \in \text{dom}(\sigma)$ then $P\sigma = P$;
  - else $P\sigma = \beta x.(Q\sigma)$.

Using substitutions, two formulas can become syntactically equal through instantiating their free variable(s) with the corresponding term(s). As such, if for two formulas P and Q there exists a substitution σ such that $P\sigma = Q\sigma$, then these two formulas are said *unifiable* with σ being their *unifier*. It is however necessary to be careful when applying a substitution upon a formula, so as not to accidentally have a free variable wrongly interact with a bound variable carrying the same name and thus also becoming bound when substituting. As such, an α-conversion of the substituted formula may have to be realised. Two formulas are said α-equivalent or α-convertible 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.

Proofs. Syntactically reasoning over formulas is called a *derivation*. Such derivation involves applying rules of a particular logical system to rewrite a formula by breaking its connectors or instantiating its quantified variables. For example, when reasoning over $P \wedge Q$, it is clear that P and Q should hold for $P \wedge Q$ to hold. As such, if applying a logical rule on a formula F yields a formula F' then one step of derivation is denoted $F \longrightarrow F'$ with $\longrightarrow^*$ denoting the reflexive and transitive closure of $\longrightarrow$. A derivation stops when an *axiomatic* rule, often denoted $\odot$, is applied and a *formal proof* of a formula F is a derivation such that $F \longrightarrow^* \odot$. It is important to note that derivations only apply over closed formulas. If there exists a formal proof of a formula F, it means that F is a *logical theorem* and is simply called *theorem* or *valid*. On the contrary, if there exists a formal proof of $\neg F$, it means that F is *countersatisfiable*. The derivation steps forming a proof depends on the system used, and some of those will be presented in more details in the next section.

2.2 Some Proof Systems: Focus on the Tableaux Method and GS3

Since Hilbert and his *Entscheidungsproblem*[1], proof systems to automatically and syntactically reason over formulas have been developed. Two disjoints categories [33] have emerged as a result of the researches led — sequent-based systems [24, 18] and resolution-based systems [32]. The former relies on deriving a proof by keeping the syntactic integrity of a formula,

[1] Hilbert's decision problem: is there an algorithm which decides if a formula is valid.

while the later simplifies the formulas by preprocessing them to allow an efficient reasoning. A sequent-like system, by keeping the syntactic integrity of formulas, has the advantage of enabling the production of a human-readable and machine-checkable proof, advantage which is lost in favor to computational efficiency by resolution systems due to the formulas' preprocessing. This section focuses on presenting the two sequent-based systems used throughout this document: a variant of the method of analytic tableaux, first introduced by Beth and Hittinka [7] and the Gentzen-Schütte calculus, denoted GS3 [38].

Free-Variable Tableaux. Free-variable tableaux is a variant of the original method introduced in [21] which allows the use of free variables instead of directly instantiating with ground terms in order to improve the performances of automated theorem provers. It is based upon a *refutational* calculus, i.e., given a set of formulas Γ called hypotheses and a conjecture F, a derivation in this method involves proving that the conjunction of Γ and $\neg F$ is countersatisfiable. The calculus is *deductive*, i.e., reads *top to bottom* and is presented in Figure 1. It is composed of α–, γ– and δ–rules that extend a branch with one formula, β–rules that divide a branch by extending it with two formulas and $\odot$–rules that close a branch.

γ–rules deal with universally quantified formulas, yielding a formula where the bound variable has been substituted by a *fresh* free variable (i.e., by a free variable that does not yet appear in the proof). In essence, free variables are not part of a proof and are instead used as a placeholder while waiting for an instantiation that is usually done upon finding a contradiction, i.e., a $\odot$–rule. A non-trivial closure rule, i.e., a $\odot$-rule involving actual formulas and neither $\top$ nor $\bot$, yields such a substitution $[X_1 \mapsto t_1, \ldots, X_n \mapsto t_n]$ denoted σ when there exists two formulas P and $\neg Q$ unifiable using σ, i.e., when $P\sigma = (\neg Q)\sigma$.

δ–rules deal with existentially quantified formulas, instantiating them on the fly using a Skolem function symbol, i.e., a *fresh* symbol which does not appear in the language of the branch. In the rules of Figure 1, `sko` is a meta function which takes all the free variables of the branch as parameters and returns a Skolem (functionnal) symbol parameterised by those very free variables. This method is called *Skolemisation* and is used to ensure freshness of the Skolem symbol independently of variable instantiation.

When all the branches of a tableau are closed by $\odot$–rules, the tableau is also said *closed*: the formula F is a theorem under the hypotheses Γ. A standard tableaux proof is shown in Figure 3a.

GS3. The Gentzen-Schütte calculus, presented in Figure 2 and conveniently arranged in a way that mirrors the tableaux rules of Figure 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. 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. Indeed, *ex falso sequitur quodlibet*[2], so if a contradiction is found in the hypotheses, then a branch can be closed. Furthermore, *contraction* (i.e., duplication) is implicitly executed upon an hypothesis once it is processed. As such, the tableaux method sticks closely to this calculus, where all the processed formulas are also kept in the branch. One important difference to note between both methods are the first-order rules, i.e., the rules involving quantifiers. GS3 is a non-automated focused proof-system, so no free variables are involved during the proof-search, thus instantiating variables to *known* terms when dealing with γ–equivalent rules and Skolemising with *fresh* constants. This system is closely related to the usual systems implemented in interactive theorem provers [6, 31, 17] and can thus easily be embedded inside such tools, as Subsec. 4.1 will show. The main properties that those systems share is the way of dealing with quantifiers and, as such, a translation of Skolemised formulas has to be devised in order to formally certify tableaux proofs.

2.3 From Tableaux to GS3, a Deskolemisation Problem

Problematic. The standard free-variable tableaux calculus uses *outer Skolemisation* for δ–rules, as can be remarked in Figure 3a. This strategy makes the final tableau totally equivalent to original tableaux [21] 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). However, to optimise tableaux proofs, Skolemisation strategies have been widely studied, offering δ^+–, δ^{+^+}–, δ^*–, δ^{*^*}– and δ^ε–rules [25, 5, 3, 12, 22], where the most optimised strategies δ^ε and δ^{*^*} are shown to yield

[2] From falsehood, anything follows.

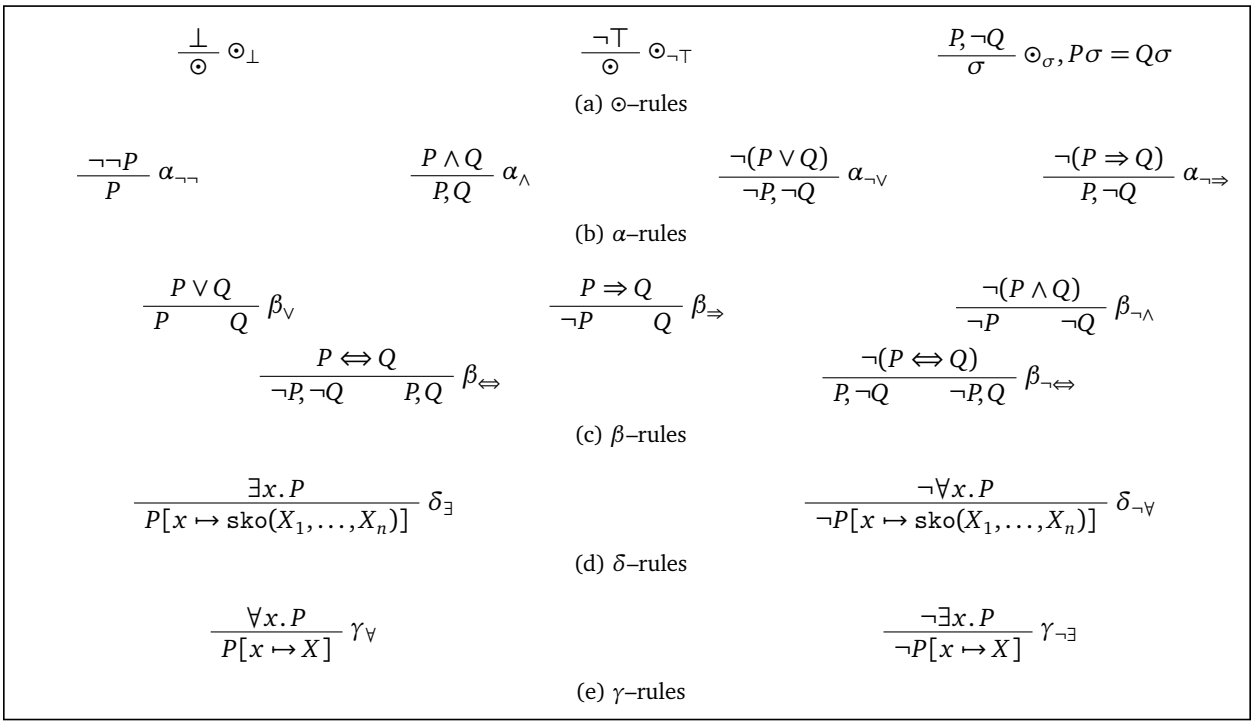


Figure 1: Rules of the Free-Variable Tableaux Calculus.

proofs shorter (for a number n of branches) by a factor of 2^{2^n} compared to standard Skolemisation. This is a great boon for proof-search procedures, but it induces problems when translating tableaux proof to certify them.

For instance, a proof in the δ^+–rules (*inner* Skolemisation rules), the weakest amelioration over standard tableaux, is developed in Figure 3b, and whereas no branch is created in this proof, it is already shorter than its outer-Skolemisation counterpart in a way that makes the proof not readily translatable into a GS3 sequent. Figure 4 is an attempt of a naive translation which fails when the rule denoted ($\star$) is applied. Indeed, recall that when Skolemising, the constant yielded needs to be *fresh* which is not the case here as c is introduced by the $\neg\exists$ rule. If the constant is made fresh, i.e., if the rule $\neg\forall$ yields c', then the axiomatic rule can not be called, as the closure in the tableaux proof is found between $D(c)$ and $\neg D(c)$ when the GS3 sequent's node will be labelled by $D(c)$ and $\neg D(c')$. To certify tableaux proofs with smart Skolemisation strategies, a translation into GS3 sequents thus needs to be devised.

Focus of the Work. During this internship, the focus has been placed upon δ^+– and δ^{+^+}– rules. The former optimises standard outer Skolemisation by solely keeping the free-variables of the Skolemised formula as arguments, which ensure a more *local* solution and an exponential gain in proof-search size (in this document, the size of a proof will always refer to the number of branches of said proof) over outer Skolemisation. This Skolemisation strategy is usually referred to as *inner* Skolemisation in the literature. The later is called *pre-inner* Skolemisation, and as its name suggests, it builds over *inner* Skolemisation 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.

State-of-the-Art Solutions. Translating tableaux proofs into GS3 sequents appear as early as 1987 for the connection tableaux [8]. These tableaux are a restriction of the original method, where formulas are preprocessed to apply an heuristic during the branches' exploration. Years later, [2, 4] have shown that deskolemising functions in inner Skolemisation results in a substantial increase of the proof size. [20]

$$\frac{}{\Delta, \bot \vdash}\ \text{ax} \qquad \frac{}{\Delta, \neg\top \vdash}\ \text{ax} \qquad \frac{}{\Delta, P, \neg P \vdash}\ \text{ax} \qquad \frac{\Delta \vdash}{\Delta, P \vdash}\ \text{w}$$

(a) Structural and Axiomatic Rules.

$$\frac{\Delta, \neg\neg P, P \vdash}{\Delta, \neg\neg P \vdash}\ \neg_\neg \qquad \frac{\Delta, P \wedge Q, P, Q \vdash}{\Delta, P \wedge Q \vdash}\ \wedge \qquad \frac{\Delta, \neg(P \vee Q), \neg P, \neg Q \vdash}{\Delta, \neg(P \vee Q) \vdash}\ \neg_\vee \qquad \frac{\Delta, \neg(P \Rightarrow Q), P, \neg Q \vdash}{\Delta, \neg(P \Rightarrow Q) \vdash}\ \neg_\Rightarrow$$

(b) Non-Branching Propositional Rules.

$$\frac{\Delta, P \vee Q, P \vdash \qquad \Delta, P \vee Q, Q \vdash}{\Delta, P \vee Q \vdash}\ \vee \qquad \frac{\Delta, P \Rightarrow Q, \neg P \vdash \qquad \Delta, P \Rightarrow Q, Q \vdash}{\Delta, P \Rightarrow Q \vdash}\ \Rightarrow$$

$$\frac{\Delta, \neg(P \wedge Q), \neg P \vdash \qquad \Delta, \neg(P \wedge Q), \neg Q \vdash}{\Delta, \neg(P \wedge Q) \vdash}\ \neg_\wedge$$

$$\frac{\Delta, P \Leftrightarrow Q, \neg P, \neg Q \vdash \qquad \Delta, P \Leftrightarrow Q, P, Q \vdash}{\Delta, P \Leftrightarrow Q \vdash}\ \Leftrightarrow \qquad \frac{\Delta, \neg(P \Leftrightarrow Q), P, \neg Q \vdash \qquad \Delta, \neg(P \Leftrightarrow Q), \neg P, Q \vdash}{\Delta, \neg(P \Leftrightarrow Q) \vdash}\ \neg_\Leftrightarrow$$

(c) Branching Propositional Rules.

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

(d) Skolemisation Rules, where c is a Fresh Constant.

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

(e) Instantiation Rules, where t is a Ground Term.

Figure 2: Rules of the GS3 Calculus.

also proposes a framework of proof deskolemisation, but is aimed towards the resolution method and as such does not conserve the syntactic integrity of the formula. Proof deskolemising for outer Skolemisation in others sequent-like systems has also been explored in [13] and [30]. Deskolemisation of δ^ε–rules have been implemented in [9], 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. Finally, [10] proposes a theoretic yet generic method to translate standard tableaux into GS3 sequents. Furthermore, as far as the author knows, no work has been done yet on deskolemising proofs using pre-inner Skolemisation rules.

2.4 Discussion and Objectives

The only solution known by the author for standard tableaux as of yet, i.e., the one presented in [10], is highly theoretic and lacks in-practice efficiency, as the exponential bound is attained for every translated proof. The objectives of the internship is thus twofold — (i) propose an automated procedure to efficiently translate tableaux proofs using δ^+ and δ^{+^+} rules into GS3 sequents and (ii) add the optimised Skolemisation strategies in Goéland [11], an automated tableaux-based theorem prover, and implement the translation algorithm. Once implemented, the objective is to embed GS3 sequents into a proof assistant, the first target being Coq, and certify Goéland's proofs by outputting a Coq-compatible proof and thus experimentally validate the approach. The genericity of the method should allow further certification, with the end goal being the Dedukti [1] framework, so as to endow Goéland with an inter-operable proof output.

3 A Deskolemisation Algorithm

Use of on-the-fly optimised Skolemisation strategies in a tableaux proof, such as δ^+ or δ^{+^+} rules and their equivalent, induce non-trivial translations of such proofs into GS3 sequents as the former's number of branches is (theoretically) shorter by an exponential factor over the later's. This gain does not bode well for the translation as it means that it should create an exponential number of branches to solve the ordering problem faced, and that naive means such as topologically sorting the rules applied by their dependencies fail (as this

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

(a) Outer-Skolemisation Tableau.

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

(b) Inner-Skolemisation Tableau.

Figure 3: Proof of the Drinker Paradox in Outer and Inner Skolemisation.

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

Figure 4: Incorrect Proof Yielded by a Naive Translation of an Inner-Skolemisation Tableaux Proof.

kind of solution uses linear-time to compute). The goal of this section is to propose an easily-generalisable — generalisable in the sense of different Skolemisation strategies or even different logics — algorithm to automatically translate a tableaux proof in a valid GS3 sequent. Hence, it starts by defining such an algorithm for δ^{+}–rules in Subsec. 3.1 and prove its soundness in Subsec. 3.2. Furthermore, we have taken the first step for the validation of the method's genericity by extending the algorithm to $\delta^{+^{+}}$–rules in Subsec. 3.3.

3.1 Presentation of the Algorithm

As it has been mentioned in the previous section and highlighted in Figure 3b, a tableau proof is sound even if a free variable is instantiated by a term which 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 Skolemisation 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 [10] and which should gently generalise to other Skolemisation strategies, not relying on a syntactic preprocessing of formulas as in [8].

Inspiration. The idea behind the algorithm of [10] 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. Then, the sequent is made to forget everything (by, in practice, weakening the relevant formulas) except the very formula D and the initial formula to prove. After the weakening process, the sequent is grown back by grafting the pre-weakened proof tree. In effect, this processing arranges the sequent to apply first and foremost every δ^{+}–rule needed in a branch before applying any other rule. It is achieved by this grow, weaken and graft strategy. As it happens, Figure 5 is an illustration of this strategy, and coincides with the more efficient version of the algorithm that will be subsequently presented.

Preliminary Definitions. The algorithm presented in the last paragraph is pathologically inefficient due to the weakening-and-grafting strategy incurred when seeing a δ^{+}–rule. As proofs in inner-Skolemisation 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 3.1: 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 $F \longrightarrow F'$ 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:
>
> $$\Delta(D) = \{F \in \Gamma_\gamma \mid F \longrightarrow F' \wedge \exists \omega. F'_{|\omega} = \delta_D\}$$

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 amount of formulas weakened to a minimum, this set must then be smartly extended by adding solely the formulas *descended* from the formulas in $\Delta(D)$ that have an occurrence of δ_D.

Definition 3.2: Descendance

> Let F be a formula of a leaf $\mathfrak{f}$ of a tableaux. Another formula $G \in \mathfrak{f}$ is said to be *descending* from F if and only if $F \longrightarrow F_1 \longrightarrow \ldots \longrightarrow F_n$ and there exists $k \leqslant n$ 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:
>
> $$\Lambda(F) = \{G \in \mathfrak{f} \mid F \longrightarrow^* G \wedge \exists \omega. G_{|\omega} = \delta_D\}$$

Algorithm. The main ideas behind the new algorithm are the same as the ones behind the previous algorithm, i.e., follow the tableaux proof by seamlessly applying the GS3 rule corresponding to the tableaux rules while the later 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 is defined as follows.

- While the rule applied is not a δ^+–rule, apply the GS3 rule corresponding to the tableaux rule and mark the formula on which it is applied (Figure 5a).
- Let D be the formula on which the δ^+–rule is applied.
- 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 5b).
- Apply the δ^+–rule on D (first step of Figure 5c).
- Apply back the rules recorded in $\mathscr{R}$ (last two steps of the Figure 5c)[3].
- Repeat while a rule is applied in the corresponding tableaux leaf.

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 3.3: No Self Dependency

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

Proof: See Appendix A.1. □

The rest of this section focuses on proving the soundness of the algorithm that has just been presented. It starts by proving soundness for inner Skolemisation, and it then extends the proof to δ^{+^+}–rules. The proofs are implicitly a part of the algorithm, as technical details that could arise when implementing it are featured inside.

3.2 Soundness of the Translation over Inner Skolemisation

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 tableaux proof. The function's domain will be total, thus associating every leaf of a valid sequent to a leaf of the

[3] This step requires some technical details, subsequently developed in the soundness proof.

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

(a) First Steps of the Proof.

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

(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) Skolemisation, Applying Back $\mathscr{R}$'s Rules and Finalisation.

Figure 5: Sound Translation of the Drinker δ^+ Proof in GS3 Using the Algorithm.

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.

Definition 3.4: Tableau Proof

> Let T be a tableau with F as its root formula, closed by a substitution σ. Then $T\sigma$ is the *tableau proof* of F. In this document, it will often be denoted T somewhat imprecisely.

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 correspond each leaf of the under-construction sequent 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 3.5: Initial Part

> Let $T\sigma$ be a tableau proof. T_0 is an *initial part* of $T\sigma$ if and only if T_0 and $T\sigma$ share the same root, the same rule is applied on this very root and all the children of T_0 are *initial parts* of the corresponding children in $T\sigma$.

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 3.6: 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 $\mathfrak{f} \in \pi$ associates a leaf in T such that $\mathfrak{f} \supseteq \mu(\mathfrak{f})$.

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 3.7: Soundness

> Let T be a tableau proof of a formula F. Then the algorithm of Subsec. 3.1 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 on D. Furthermore, all descendants of a formula depending from 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 Skolemisation 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 3.8 on T (which is in fact an initial part of itself). Then, for every leaf $\mathfrak{f} \in \pi'$, $\mathfrak{f} \supseteq \mu(\mathfrak{f})$. As $\mu(\mathfrak{f})$ contains a contradiction, $\mathfrak{f}$ 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 3.8 is to step-by-step build a mapping between the sequent generated by the algorithm and a well-chosen tableaux 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 tableaux proof quite literally, thus yielding a clear mapping between both proofs. However, it is not so clear for δ^+–rule and Lemma 3.10 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 3.8: 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: By induction on the number of rules applied in T_0. Most cases are trivial and done in Appendix A.2. The δ^+–rules case is detailed in Lemma 3.10.

□

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 so as to make correspond the work-in-progress one to the one yielded by the induction hypothesis of Lemma 3.8, 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

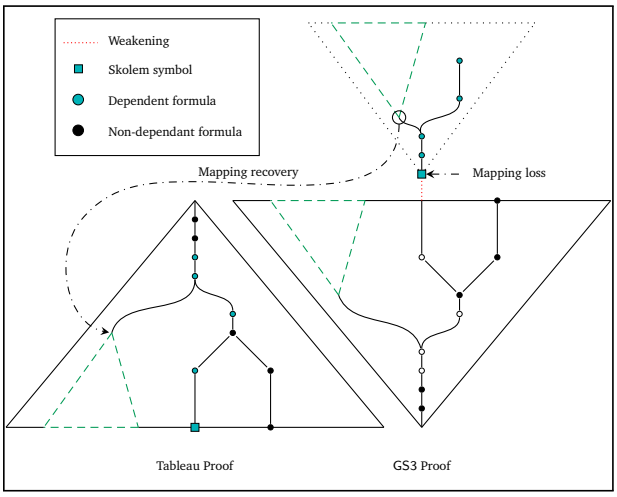


Figure 6: Mapping Conservation After Reapplication of a β–Rule.

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 3.9: Subsumed Initial Parts

> Let π be a sequent, $\mathfrak{B}$ be a branch of this sequent and E a set of formulas. The subsumed initial parts of π by E is denoted $\Pi(\pi, \mathfrak{B}, E)$ and contains all the initial parts of π such that the set labelling the leaf of the branch $\mathfrak{B}$ 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 3.10: δ^+–Rules Mapping Conservation

> Let $\mu_1 : \mathcal{L}(\pi_1) \to \mathcal{L}(T_1)$ be a mapping, $\mathfrak{f}$ be an open leaf of T_1 such that the next rule applied on $\mathfrak{f}$ is a δ^+–rule on a formula D, π_0 be the GS3 sequent after execution of the routine on π_1 and $\mu_1^{-1}(\mathfrak{f})$ 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 of itself (by Lemma 3.3) and that, by transitivity, it can not depend on any of its dependencies. Let π_0^0 be π_1 where the branch $\mathfrak{B}$ carrying $\mu^{-1}(\mathfrak{f})$ is extended by following the algorithm before applying back the rules, i.e., where all formulas of $\Lambda(D)$ are weakened and the Skolemisation rule has been applied on D, generating D'. As such, let Π_1^0 be $\Pi(\pi_1, \mathfrak{B}, \mu^{-1}(\mathfrak{f}))$, i.e., the subsumed initial parts of π_1 over $\mathfrak{B}$ and the set $\mu^{-1}(\mathfrak{f})$. Π_1^0 can be totally ordered by inclusion and we can thus take π_1^0 to be $\max(\Pi_1^0)$, with $\mathfrak{b}$ being its leaf on $\mathfrak{B}$. 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 $\mathfrak{n}$ of π_0^0:

$$\lambda_1^0(\mathfrak{n}) = \begin{cases} \mathfrak{b} & \text{if } \mathfrak{n} \text{ is } (\mu^{-1}(\mathfrak{f}) \setminus \Lambda(D)) \cup \{D'\} \\ \mathfrak{n} & \text{otherwise} \end{cases}$$

We then construct by induction over the number n of formulas that have been weakened a family of mappings $(\lambda_1^i)_{i \leqslant n}$, initial parts $(\pi_1^i)_{i \leqslant n}$ and sequents $(\pi_0^i)_{i \leqslant n}$ 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 $\mathfrak{B}$ has been extended with φ, yielding the leaf $\mathfrak{f}'$, Π_1^k is $\Pi(\pi_0^k, \mathfrak{B}, \mathfrak{f}')$ and thus $\pi_1^k = \max(\Pi_1^k)$. Let $\mathfrak{b}$ be the leaf of $\mathfrak{B}$ in π_1^k. Then λ_1^k is defined as follows for every leaf $\mathfrak{n}$ of π_0^k:

$$\lambda_1^k(\mathfrak{n}) = \begin{cases} \mathfrak{b} & \text{if } \mathfrak{n} \text{ is } \mathfrak{f}' \\ \lambda_1^{k-1}(\mathfrak{n}) & \text{otherwise} \end{cases}$$

  - If r is a β–rule which produces φ_1 and φ_2, then let us suppose without loss of generality that $\varphi_1 \in \mu_1^{-1}(\mathfrak{f})$. Let us define π_2 to be

π_0^{k-1} where $\mathfrak{f}'$ the leaf of $\mathfrak{B}$ has been extended into two leaves $\mathfrak{f}_1$ ($\ni \varphi_1$) and $\mathfrak{f}_2$ ($\ni \varphi_2$), Π_1^k is $\Pi(\pi_2, \mathfrak{B}, \mathfrak{f}_1)$ and thus $\pi_1^k = \max(\Pi_1^k)$. As such, let $\mathfrak{b}$ be the parent of $\mathfrak{B}$'s leaf in π_1^k. The mapping of $\mathfrak{f}_1$ is straightforward as it is simply $\mathfrak{b} \cup \{\varphi_1\}$. However, selecting π_0^k and $\mathfrak{f}_2$'s mapping is not. Indeed, the node $\mathfrak{b} \cup \{\varphi_2\}$ 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 $\mathfrak{b} \cup \{\varphi_2\}$ in π_1^k. π_0^k is defined to be π_2 where the subsequent tree that is rooted at $\mathfrak{f}_2$ becomes π_3 (therefore, if π_3 is rooted at $\mathfrak{f}_3$ then $\mathfrak{f}_2 \supseteq \mathfrak{f}_3$) as illustrated in Figure 6. Then λ_1^k is defined as follows for every leaf $\mathfrak{n}$ of π_0^k:

$$\lambda_1^k(\mathfrak{n}) = \begin{cases} \mathfrak{b} \cup \{\varphi_1\} & \text{if } \mathfrak{n} \text{ is } \mathfrak{f}_1 \\ \mathfrak{n} & \text{if } \mathfrak{n} \text{ is a leaf of } \pi_3 \\ \lambda_1^{k-1}(\mathfrak{n}) & \text{otherwise} \end{cases}$$

- In case 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 $({\pi'}_0^i)_{i \leqslant m}$, $({\pi'}_1^i)_{i \leqslant m}$ and $({\lambda'}_1^i)_{i \leqslant m}$ and thus giving $\pi_1^k = {\pi'}_1^m$, $\pi_0^k = {\pi'}_0^m$ and $\lambda_1^k = {\lambda'}_1^m$.

Finally, π_0^n is the sequent where all the rules have been applied back and thus is π_0 and if $\mathfrak{b}$ is its branch $\mathfrak{B}$'s leaf, then $\mathfrak{b} \supseteq \mu_1^{-1}(\mathfrak{f})$ and π_1^n is π_1. As such, as $\lambda_1^n(\mathfrak{b}) \subseteq \mathfrak{b}$ and $\lambda_1^n(\mathfrak{b}) \in \mathcal{L}(\pi_1)$ then $\mathfrak{b} \supseteq \mu_1(\lambda_1^n(\mathfrak{b}))$ and μ_1 can be extended for every leaf $\mathfrak{n}$ of π_0 as follows to yield μ_0:

$$\mu_0(\mathfrak{n}) = \begin{cases} \mu_1(\lambda_1^n(\mathfrak{n})) \cup \{D'\} & \text{if } \mathfrak{n} \text{ is } \mathfrak{b} \\ \mu_1(\lambda_1^n(\mathfrak{n})) & \text{otherwise} \end{cases}$$

□

3.3 Extending Deskolemisation to δ^{+^+}

Most of the problems faced when translating δ^{+^+} tableaux proofs to GS3 sequents are properly managed by the translation algorithm. Indeed, as those rules build over inner Skolemisation, the ordering of the rules applied are naturally taken into account when deskolemising. However, on-the-fly Skolemisation 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. In turn, this change unknowingly makes the algorithm proposed in Subsec. 3.1 diverge, as shown in Figure 7.

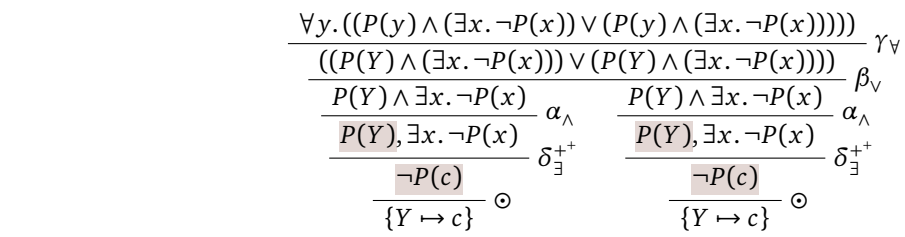
Figure 7a is the closed tableaux of a (somewhat unnatural) formula, where both δ^{+^+}–rules give the same symbol and use it to close their respective branch, having the same final substitution. The translation of this proof by the deskolemisation algorithm is illustrated in Figure 7b. As the root formula Γ depends of the δ-formulas found in the two branches, and as they generate the exact same Skolem symbol, weakening the formulas of the other branch when applying back a β–rule (or, independently, weakening the dependent formulas before a δ^{+^+}–rules) to graft the right-hand side of the proof induces a divergence: the δ–rule is also applied on the other side of the proof and thus a grafting of this side is subsequently needed, leading to an infinite loop between the two branches.

Intuitively, when downgrading a δ^{+^+} proof to a δ^+ proof, there are two cases when applying a Skolemisation 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 8 shows the proof in δ^+, which is the counterpart of the δ^{+^+} proof of Figure 7a. In fact, such a proof is eerily similar to the GS3 sequent given by the translation algorithm of Figure 7b, minus the fact that no δ–rules are applied back after reintroducing as the needed-for-closure formula ($\neg P(c_1)$) is not weakened when branching.

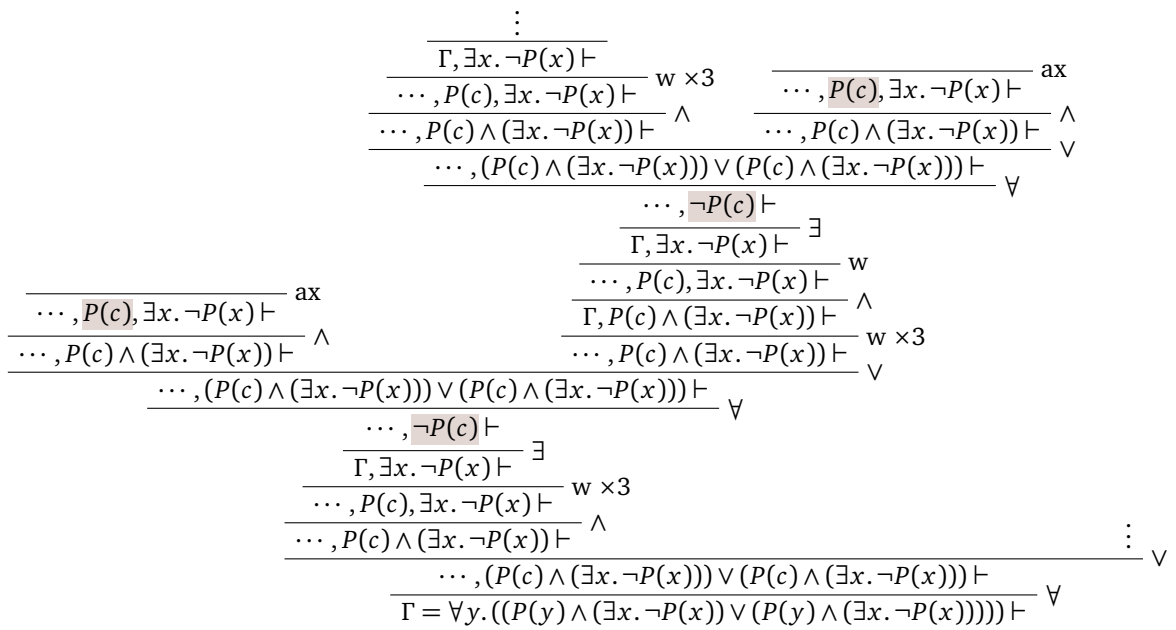
As such, extending the translation algorithm to δ^{+^+} proofs is fairly straightforward: it suffices to avoid weakening the Skolemised formulas and reapplying a δ^{+^+}–rule over a previously Skolemised formula. Intuitively, two cases can then happen — (i) the non-weakened Skolemised formula is useless 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 formalise this intuition, the proof of Lemma 3.10 should be reworked for β–rules to properly select the sequent over which both resulting branches map to.

Lemma 3.11: δ^{+^+}–Rules Mapping Conservation

> Let $\mu_1 : \mathcal{L}(\pi_1) \to \mathcal{L}(T_1)$ be a mapping, $\mathfrak{f}$ be an open leaf of T_1 such that the next rule applied on $\mathfrak{f}$ is a δ^{+^+}–rule on a formula D, π_0 be the GS3 sequent after execution of the routine on π_1 and $\mu_1^{-1}(\mathfrak{f})$ and T_0 the tableau T_1 after application



(a) Tableaux Proof Yielding the Same δ^{+^+}-Symbol.



(b) A Diverging Translation Using The Algorithm.

Figure 7: Formula that makes the Translation Algorithm Diverge.

> 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 $\mu_1^{-1}(\mathfrak{f})$ is considered to be on the branch $\mathfrak{B}$ 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 $\varphi_1 \in \mu_1^{-1}(\mathfrak{f})$. As such, let π_2 be π_0^{k-1} where the leaf of $\mathfrak{B}$ in π_0^{k-1} has been extended in two leaves $\mathfrak{f}_1$ ($\ni \varphi_1$) and $\mathfrak{f}_2$ ($\ni \varphi_2$). Let π_1^k be $\max(\Pi(\pi_2, \mathfrak{B}, \mathfrak{f}_1))$ and $\mathfrak{f}'$ be $\lambda_1^{k-1}(\mathfrak{f}_1 \setminus \{\varphi_1\})$, i.e., the parent node of the leaf of $\mathfrak{B}$ in π_1^k. Recall that $\mathfrak{f}' \cup \{\varphi_1\}$ is a leaf of π_2 as well as π_1^k so the mapping is straightforward. On the other hand, $\mathfrak{f}' \cup \{\varphi_2\}$ might not be a leaf of π_1^k, and as such let π_3 be the sequent starting at the node $\lambda_1^{k-1}(\mathfrak{f}') \cup \{\varphi_2\}$ in π_1^k. Then, let π_3' be π_3 where all the nodes have been aug-

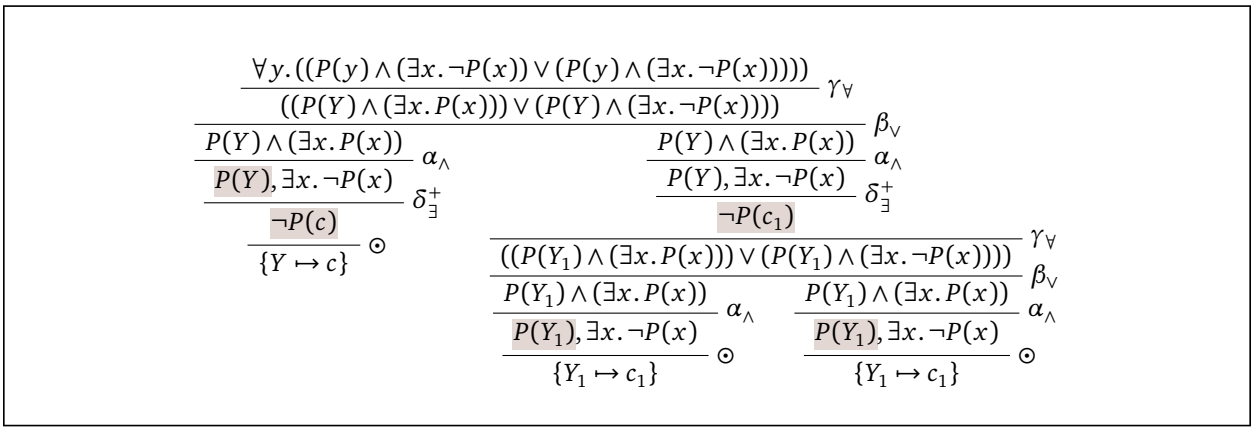


Figure 8: Translation of the δ^{+^+} Proof to the δ^+ Proof.

mented with D', i.e., if $\mathfrak{n}$ is a node of π_3, then $\mathfrak{n} \cup \{D'\}$ is a node of π'_3. Thus it suffices to define π^k_0 to be π_2 where $\mathfrak{f}_2$ is replaced by grafting π'_3, i.e., where the sub-tree $\mathfrak{f}_2$ is replaced by the tree π'_3. As such, $\lambda^k_1 : \mathcal{L}(\pi^k_0) \to \mathcal{L}(\pi^k_1)$ can be defined as follows for every leaf $\mathfrak{b}$ of π^k_0:

$$\lambda^k_1(\mathfrak{b}) = \begin{cases} \mathfrak{f}' \cup \{\varphi_1\} & \text{if } \mathfrak{b} \text{ is } \mathfrak{f}_1 \\ \mathfrak{b} \setminus \{D'\} & \text{if } \mathfrak{b} \in \pi'_3 \text{ and the corresponding} \\ & \text{node is not labelled with } D' \text{ in } \pi_3 \\ \mathfrak{b} & \text{if } \mathfrak{b} \in \pi_3 \\ \lambda^{k-1}_1(\mathfrak{b}) & \text{otherwise} \end{cases}$$

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., $\mathfrak{b} \supseteq \lambda^k_1(\mathfrak{b})$) 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 which can serve as a solid basis for deskolemising. The author is hopeful that it generalises well for other logics and Skolemisation strategies, and that only slight modifications will be needed when applying back rules to lift the method to all the different δ–rules available.

4 Implementation and Performances Evaluation

Implementing the theoretic solution proposed in Section 3 has been a key part of the internship and has served two purposes: (i) speed-up Goéland's proof-search by using smarter Skolemisation strategies and (ii) prove that the algorithm has real-world value by deskolemising state-of-the-art proofs and show that translation is, in practice, cheap, even though it is, in theory, exponential. As such, an overview of the implementation of all the mechanisms needed to certify a proof is given in Subsec. 4.1. In turn, Subsec. 4.2 exhibits the results of the method by giving a key metric over different Skolemisation strategies and details the testing strategy used to obtain those results.

4.1 A Framework for Proof Certification: an Implementation in Goéland

This section exposes the implementation work that has been done during the internship. It is split into two parts, starting upstream by presenting the updates needed in the proof-search procedure to support both more Skolemisation strategies and a proof output and ending downstream by explaining how the translation to Coq has been set up in Goéland, detailing the in-between GS3 framework established and thus giving a complete overview of a proof certification's process.

Updates to the Proof-Search. The core of Goéland, its proof-search procedure, has needed two updates during the internship. One minor taking form in an implementation of different Skolemisation strategies and one major in its proof output.

The former has been easily managed by adding methods in the interface of a formula to store the internal free-variables present in it. It may sound like a redundancy 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 on the whole tableau. Thus, free-variables are lost during this process. As such, this storage becomes necessary to pass the right arguments when Skolemising. Furthermore, two flags have been set up to change the Skolemisation mode: `-inner` which activates δ^+–rules and `-preinner` that makes use of δ^{+^+}–rules.

The later, the proof output's update, has needed a more thorough investigation and a dive into the proof-search procedure is needed. Goéland is a concurrent automated theorem prover, launching its proof-search procedure in parallel on each child when applying a β–rule. The children search by themselves until they find a contradiction and, potentially, a substitution. If a substitution involving free-variables introduced by the parent or a node higher in the tree is indeed found by any child, then the parent has to launch a reconciliation algorithm between all of them. In general, this algorithm has to select a child, apply its returned substitution and launch back the proof-search on the others. However, if all the children have replied back a *compatible* substitution, i.e., when the composition of all the children's substitutions is a functional relation, then the parent considers that they agree and can thus propagate the substitution upwards without informing its children. This, in turn, causes a substitution-induced bug in the proof output. Indeed, as can be seen in Figure 9, the parent selects the *compatible* substitution $\{X \mapsto a\}$ and thus, both A and B choose an arbitrary substitution to close their proof. Then, A can select the substitution containing $\{X \mapsto b\}$, which, in turn, induces an inconsistency in the proof. As reliable proofs are needed in an automated theorem proving tool, a mechanism yielding sound proofs (highlighted in the figure) has been implemented: the global unifier. This name is fitting, as it builds an unifier on-the-fly for the whole free-variable tableau. This mechanism offers *at least* one unifier that closes it when the proof-search procedure ends, thus outputting a sound proof.

Translating the Proof. The sound proof output of Goéland is a mean towards the goal of outputting certified proofs. A translation must then take place in order to transform Goéland's tableaux proof to a machine-checkable system. We have chosen to implement this translation using two layers: the deskolemisation layer and the transformation layer. The former transforms a tableaux proof to a deskolemised GS3 sequent. It is a straightforward implementation of the algorithm given in Section 3 and can be found in the Goéland's public repository[4]. The advantage of this deskolemisation step is that it offers a sequent proof that is easily checkable by any proof-assistant, as GS3 can be embed in most of such tools. For instance, a Coq embedding of GS3 has been implemented during this internship and can be found in Appendix B. Most of the file consists solely of lemmas that represent the GS3 rules, which can be used almost instantly. As such, to finish the translation work, a one-to-one mapping between the GS3 proof and the Coq embedding has to be applied to finally produce a proof that can be automatically certified. Coq is the first proof assistant that has been chosen due to the author familiarity with this system, but the layer of abstraction added by the GS3 sequent allows to easily translate Goéland's proofs into the language of any proof-assistant. It is planned to implement a Goéland-to-Dedukti translation to validate the genericity and re-usability of the method.

4.2 Testing the Algorithm: Towards an Efficient Translation for Different Skolemisation Strategies

Since 1993, benchmarks around automated theorem provers have revolved around the Thousand of Problems for Theorem Provers (TPTP) problem library [36]. This library is the *de facto* reference for testing the developed tools as it has developed a standardised way to represent logical problems and features over nine thousand (first-order logic) problems, ranging from syntactic theorems all the way to industrial proof obligations. As such, we have also tested Goéland and some of its variants over a select few TPTP problems to test the proof certification algorithm. The results yielded are summarised in Table 1. This section is divided in two parts. First, the methodology to select the problems and run the benchmarks is discussed to allow reproduction of the results. Second, the result's table and its implications are explained.

[4] https://github.com/GoelandProver/Goeland, on the branch `dev/jro`, in the folder `plugins/gs3`

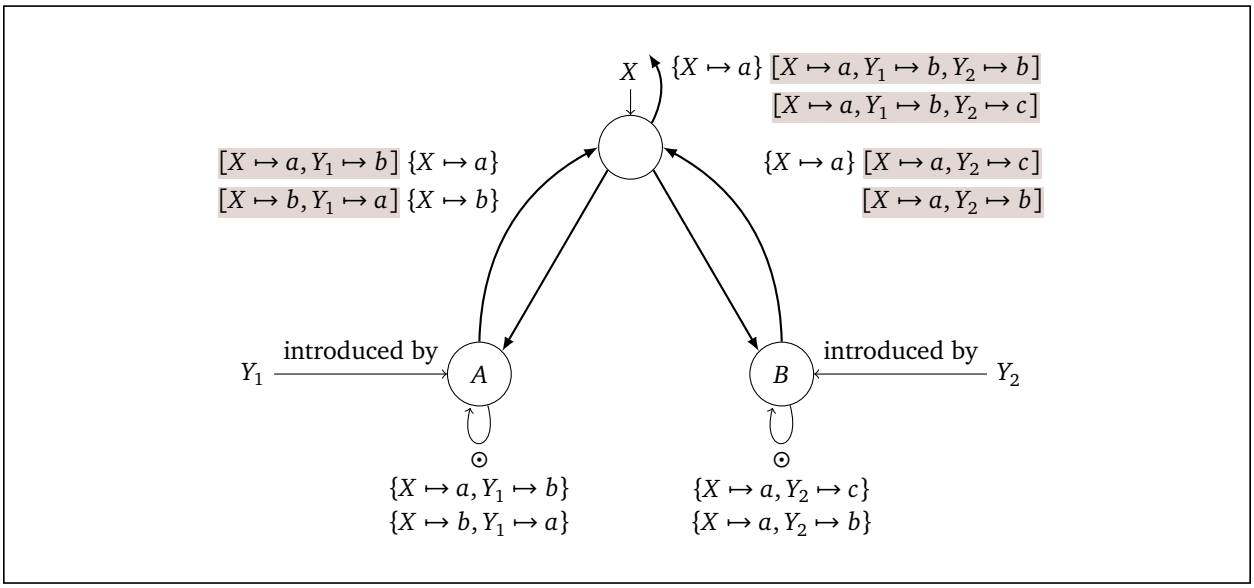


Figure 9: Global Unifier to Fix the Bug of the Proof-Search Procedure.

	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%	10.3%	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 1: Comparison Between the Different Skolemisation Strategies and their Proof-Size Increase.

Methodology. Usually, due to the CADE ATP System Competition (CASC) [37], the timeout is set at 300s when launching an automated theorem prover over a particular problem. As such, we have also adopted this practice to set state-of-the-art benchmarks for the proposed method. However, in order to test the prover over all the TPTP first-order problems using this time limit, more than a month is (in the worst case) necessary. A set of problems has thus been selected to drastically decrease the time needed to run the benchmarks. The problems selected are part of two categories: the syntactic (SYN) problems and the naive set theory (SET) problems. The former is composed of problems that are made mostly to test different aspects of a tool. It offers the insurance that if everything works properly for this category, then it should also work as expected for all the others categories. The later proposes standard reasoning over real-world problems inside a theory. Such problems are, in general, harder to prove. Nevertheless, an extension of Goéland, the deduction modulo theory (DMT) [19], allows it to reason efficiently inside the SET category and as such offers a wider range of problems to certify. Once these two categories were focused, another selection took place by launching Goéland and Goéland+DMT (Goéland with deduction modulo theory activated) on an HPC platform equipped with 28 cores (Intel Xeon E5-2680 v4 2.4 GHz) and 128 gigabytes of RAM to create a subset of the problems proved by the two variants. These subsets, together with Goéland and the benchmark script, are available online[5]. Then, the tests have been launched on a quadra-core

[5]Benchmarked problems available on https://github.com/GoelandProver/GoelandBenchmarks/ in the PROOF_CERTIFICATION folder.

(Intel Core i5-1145G7 2.6GHz) machine with 16 gigabytes of RAM 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[6]. It is important to note that, as Goéland is a parallel theorem prover, its proof-search algorithm is *non-deterministic* and as such, the results presented here may not be perfectly reproducable. Nevertheless, in any case, the percentage of problems certified should near 100%.

Insight into the Results. Table 1 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 *tableaux proof* has been output by the variant. The third column gives the percentage of tableaux proofs that have been successfully translated to Coq's proofs. The fourth and fifth columns present the size increase between the tableaux proof and the Coq's proof. The former exhibits the average size increase between both proofs while the later indicates the maximum ratio obtained in the considered subset. The results obtained are very promising as (i) every proof of all the variants have been properly certified and (ii) the average size increase between the two versions of the proof is low. Indeed, in theory, a tableaux proof is exponentially better than its GS3 counterpart in inner Skolemisation. However, for both variants featuring this Skolemisation strategy, the average and maximum size increase are low. Indeed, for the Goéland+δ^+ variant, in average, only two more branches are created during the translation. Furthermore, the maximum increase is realised 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 in average it does not even increase the proof size of one branch, and the maximum ratio is attained 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 Skolemisation 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 attained 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 Skolemisation strategy, as it behaves as expected: a one-to-one translation between the tableaux proof and the Coq proof is realised, 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. The author thus hopes that this result creates a bridge between automated theorem provers using optimised proof-search strategies and proof assistants that do not natively handle the proofs output of such strategies.

[6] Still in the previous directory.

5 Conclusion and Future Work

In summary, we have proposed a new generic algorithm to deskolemise tableaux proofs with different Skolemisation strategies by combining the ideas from [10, 21] and showed its soundness. This algorithm shows promises as, as far as the author knows, it is the first one that generalises properly to a more optimised Skolemisation strategy. As such, the author hopes that it can become a solid basis for further deskolemisation investigations.

In parallel, this algorithm has been implemented in Goéland, a tool which provides a standard tableaux-proof output, and has thus enabled the translation of its proofs to GS3, the system used in the algorithm. Then, a Coq embedding of GS3 has been built thus allowing an implementation of a translation from GS3 to Coq in Goéland.

The promising results showed in practice by the algorithm over δ^+–rules proofs has then comforted us in implementing the extensions to handle δ^{+^+}–rules and deduction modulo theory, two others proof optimisation 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 Skolemisation: δ^{ε}–, δ^{*}– and $\delta^{*^{*}}$–rules.

Nonetheless, the next goal should come in the form of proving the soundness of the method in the context of deduction modulo, as it has been implemented but no guarantees whatsoever are given for the translation to remain sound. Empirical results have however suggested that it may integrate seamlessly in the proposed framework.

Furthermore, instead of implementing a deskolemisation process in every automated theorem prover, a standard for outputting tableaux proofs could be devised to feed such proofs to a tool that implements the translation algorithm. This tool could be built inside a certified environment such as Coq and thus allow an automatic certification of tableaux proofs.

References

[1] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti : a Logical Framework based on the $\lambda\Pi$-Calculus Modulo Theory. 2016.

[2] Jeremy Avigad. Eliminating Definitions and Skolem Functions in First-Order Logic. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 139–146. IEEE Computer Society, 2001.

[3] Matthias Baaz and Christian G. Fermüller. Nonelementary Speedups between Different Versions of Tableaux. In Peter Baumgartner, Reiner Hähnle, and Joachim Posegga, editors, *TABLEAUX '95*, volume 918 of *Lecture Notes in Computer Science*, pages 217–230. Springer, 1995.

[4] Matthias Baaz, Stefan Hetzl, and Daniel Weller. On the Complexity of Proof Deskolemization. *J. Symb. Log.*, 77(2):669–686, 2012.

[5] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. The Even More Liberalized δ-rule in Free Variable Semantic Tableaux. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Computational Logic and Proof Theory*, pages 108–119, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[7] Evert Willem Beth. *Formal Methods: An Introduction to Symbolic Logic and to the Study of Effective Operations in Arithmetic and Logic*, volume 4 of *Synthese Library*. D. Reidel Pub. Co., 1970.

[8] Wolfgang Bibel. *Automated theorem proving, 2nd Edition*. Artificial intelligence. Vieweg, 1987.

[9] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs. In Nachum Dershowitz and Andrei Voronkov, editors, *LPAR 2007*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.

[10] Richard Bonichon and Olivier Hermant. A Syntactic Soundness Proof for Free-Variable Tableaux with on-the-fly Skolemization. 2013.

[11] Julie Cailler, Johann Rosain, David Delahaye, Simon Robillard, and Hinde Lilia Bouziane. Goéland: A Concurrent Tableau-Based Theorem Prover (System Description). In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning*, pages 359–368. Springer International Publishing, 2022.

[12] Domenico Cantone and Marianna Nicolosi Asmundo. A Further and Effective Liberalization of the δ-Rule in Free Variable Semantic Tableaux. In Ricardo Caferra and Gernot Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics, Selected Papers*, volume 1761 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 1998.

[13] Kaustuv Chaudhuri, Matteo Manighetti, and Dale Miller. A Proof-Theoretic Approach to Certifying Skolemization. In Assia Mahboubi and Magnus O. Myreen, editors, *CPP 2019*, pages 78–90. ACM, 2019.

[14] Alonzo Church. A Note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936.

[15] René Cori and Daniel Lascar. *Mathematical Logic: Part 1: Propositional Calculus, Boolean Algebras, Predicate Calculus, Completeness Theorems*. OUP Oxford, 2000.

[16] Haskell B. Curry, Robert Feys, and William Craig. Combinatory Logic, Volume I. *Philosophical Review*, 68(4):548–550, 1959.

[17] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015.

[18] Anatoli Degtyarev and Andrei Voronkov. The Inverse Method. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 179–272. Elsevier and MIT Press, 2001.

[19] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *J. Autom. Reason.*, 31(1):33–72, 2003.

[20] Michael Färber and Cezary Kaliszyk. No Choice: Reconstruction of First-order ATP Proofs without Skolem Functions. In Pascal Fontaine, Stephan Schulz, and Josef Urban, editors, *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning*, volume 1635 of *CEUR Workshop Proceedings*, pages 24–31. CEUR-WS.org, 2016.

[21] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, 1990.

[22] Martin Giese and Wolfgang Ahrendt. Hilbert's ϵ-Terms in Automated Theorem Proving. In Neil V. Murray, editor, *TABLEAUX '99*, volume 1617 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 1999.

[23] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[24] Reiner Hähnle. Tableaux and Related Methods. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 100–178. Elsevier and MIT Press, 2001.

[25] Reiner Hähnle and Peter H. Schmitt. The Liberalized δ-Rule in Free Variable Semantic Tableaux. *J. Autom. Reason.*, 13(2):211–221, 1994.

[26] D. Hilbert. Über das Unendliche. *Mathematische Annalen*, 95:161–190, 1926.

[27] William A. Howard. The Formulae-as-Types Notion of Construction. 1969.

[28] Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby C. Murray, and Gernot Heiser. Formally Verified Software in the Real World. *Commun. ACM*, 61(10):68–77, 2018.

[29] Adrien Marie Legendre. *Éléments de géométrie*. Firmin-Didot, 1886.

[30] Dale A. Miller. A Compact Representation of Proofs. *Stud Logica*, 46(4):347–370, 1987.

[31] Lawrence C. Paulson. Natural Deduction as Higher-Order Resolution. *J. Log. Program.*, 3(3):237–258, 1986.

[32] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):2341, jan 1965.

[33] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[34] Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel. A verified prover based on ordered resolution. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 152–165. ACM, 2019.

[35] Stephan Schulz. E - a Brainiac Theorem Prover. *AI Commun.*, 15(2-3):111–126, 2002.

[36] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure - From CNF to TH0, TPTP v6.4.0. *J. Autom. Reason.*, 59(4):483–502, 2017.

[37] Geoff Sutcliffe. The 9th IJCAR Automated Theorem Proving System Competition - CASC-J9. *AI Commun.*, 31(6):495–507, 2018.

[38] Anne Sjerp Troelstra and Helmut Schwichtenberg. *Basic proof theory, Second Edition*, volume 43 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 2000.

[39] Andrei Voronkov. The Anatomy of Vampire Implementing Bottom-up Procedures with Code Trees. *J. Autom. Reason.*, 15(2):237–265, 1995.

A Additional Proofs

This appendix regroups the proofs that were skipped in the document, as they are not needed to fully understand the work done during the internship.

A.1 No Self Dependency

Lemma 1.1:

> Let D be $\exists x.D'$ or $\neg\forall x.D'$, i.e., a δ^+–rule can be applied on 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 tableaux 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 Skolemisation, 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.

□

A.2 δ^+-Proofs Mapping

Lemma 1.2:

> 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 $\mathfrak{f}$ of T_0 which is different than the root, i.e., at least one rule r has been applied on a formula φ to yield $\mathfrak{f}$. As such, let T_1 be T_0 without the formulas of $\mathfrak{f}$ generated by its last rule and let $\mathfrak{f}'$ 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 on $\mathfrak{f}'$. Let thus π_0 be π_1 where the GS3 rule corresponding to the tableaux rule r is applied in $\mu^{-1}(\mathfrak{f}')$ on φ (which exists in this leaf, as $\mathfrak{f}' \subseteq \mu^{-1}(\mathfrak{f}')$). 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 $\mathfrak{f}'$ and thus $\mathfrak{f} = \mathfrak{f}'$.
  - r is an α– or γ–rule and in this case, $\varphi \longrightarrow \psi$. Thus, as ψ is also in $\mathfrak{f}$, the mapping $\mu_0 : \mathcal{L}(\pi_0) \to \mathcal{L}(T_0)$ is defined as follows for every leaf $\mathfrak{b}$ of π_0:

$$\mu_0(\mathfrak{b}) = \begin{cases} \mathfrak{f}' \cup \{\psi\} & \text{if } \mathfrak{b} \text{ is } \mu^{-1}(\mathfrak{f}') \cup \{\psi\} \\ \mu_1(\mathfrak{b}) & \text{otherwise} \end{cases}$$

  - r is a β–rule and in this case, $\varphi \longrightarrow \psi_1, \psi_2$. Thus, $\mathfrak{f}'$ is split into two leaves $\mathfrak{f}_1$ and $\mathfrak{f}_2$ where, without loss of generality, $\psi_1 \in \mathfrak{f}_1$ and $\psi_2 \in \mathfrak{f}_2$. The mapping $\mu_0 : \mathcal{L}(\pi_0) \to \mathcal{L}(T_0)$ is defined as follows for every

leaf $\mathfrak{b}$ of π_0:

$$\mu_0(\mathfrak{b}) = \begin{cases} \mathfrak{f}_1 & \text{if } \mathfrak{b} \text{ is } \mu^{-1}(\mathfrak{f}') \cup \{\psi_1\} \\ \mathfrak{f}_2 & \text{if } \mathfrak{b} \text{ is } \mu^{-1}(\mathfrak{f}') \cup \{\psi_2\} \\ \mu_1(\mathfrak{b}) & \text{otherwise} \end{cases}$$

- r is a δ^+–rule and then μ_0 is the mapping yielded by applying Lemma 3.10 on π_0, T_0, μ_1 and $\mathfrak{f}'$.

□

B 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_notnot : ∀ (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)) → ⊥).

(** Those lemmas are from the wrong side, which means that when proving, the rules are applied
    on 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.
```