Computer-Assisted Proof Verification for Higher-Order Automated Reasoning within the Dedukti Framework

Melanie Taprogge

University of Greifswald

A Thesis submitted for the Degree
of Master of Science

Supervised by
Prof. Dr. Alexander Steen
and Dr. Hab. Frédéric Blanqui

July 2024

Author's Note

This version of the thesis includes corrections made after the original submission. Minor typographical errors have been addressed, and a correction has been made to the encoding of the inference rule for functional extensionality, primarily affecting Sect. 5.2.1.

Abstract

Automated Theorem Proving is a core area of artificial intelligence research, focusing on the development of software that autonomously proves complex statements from given premises across various application fields. However, the diversity of output encodings of specialized systems hinders the verification of derived proofs and cooperation between provers. The Dedukti framework addresses this issue by implementing the $\lambda\Pi$-calculus modulo, enabling proofs originating from different frameworks to be expressed, combined, and checked automatically. Formal verification of fully automated higher-order logic provers remains an unmet challenge. This thesis establishes a theoretical foundation for such verification by identifying common challenges and developing general encoding strategies. Based on these strategies, a modular encoding of selected inference rules of the versatile prover Leo-III is presented. The effectiveness of this approach is empirically evaluated through the partial automation of generating proof certificates verifiable within the Dedukti framework.

Acknowledgments

The completion of this thesis would not have been possible without the invaluable support of my supervisors, Alexander Steen and Frédéric Blanqui, to whom I express my deepest gratitude. Working on this project was not only a source of great joy for me but also deepened my understanding of theoretical aspects of the topic and taught me a great deal about programming. Furthermore, it paved the way for my future work in my PhD studies. I am therefore very grateful to have had the opportunity to work on this topic and truly could not have asked for a better team of supervisors.

Alexander Steen first introduced me to the fascinating fields of logic and automated reasoning. His excellent lectures and the opportunity to gain my first experiences in scientific work as a student research assistant under his supervision were not only among the most enjoyable aspects of my studies, but have also taught me more than any other experience. He is therefore the primary reason why, despite my best intentions of becoming a theoretical ecologist when I first started this Master's degree, I *accidentally* ended up in the field of symbolic AI. In hindsight, this is a turn of events that I could not be more pleased with.

Frédéric Blanqui is at the heart of the network of computer scientists working on projects in the Dedukti framework, and the warm welcome he extended to me allowed me to connect with researchers working on similar projects, which proved to be extremely interesting and beneficial. Throughout this project, he has provided invaluable insights, guidance, and advice. His ideas and feedback have profoundly shaped this project, often leading to significant improvements and more than one complete revision for the better.

I would also like to thank the other members of Deducteam and other scientists in the field who provided me with advice. In particular, I want to thank Alessio Coltellacci, who faced similar challenges to the ones discussed in this work during his PhD project and shared his insights with me, making a valuable contribution to this work.

The companionship and support provided by my friends Markus, Julia, Hiago, Heiko, and others, as well as my family and, above all, my partner Wyatt, have been an invaluable source of encouragement. I am grateful to them both for supporting me in my work - even though some of them might have learned more about logic than they would have preferred in the process - and for the many enjoyable hours they spent with me, helping me to maintain a balance even when I got very absorbed in my work. My parents, in particular, have always encouraged me to pursue education and have been unwavering in their support. Completing this degree would not have been possible without them.

Lastly, I am grateful for the support of the Max Weber-Programm, the EuroProofNet, and the Gesellschaft für Informatik, which made it possible for me to participate in conferences, workshops, and other events.

To Sabine

Contents

1 Introduction

How can one know with certainty what conclusions follow from given premises? Philosophers have systematically approached this question since antiquity [19], giving rise to the fields of logic and reasoning. These disciplines nowadays provide frameworks to formulate chains of logical inferences unambiguously and demonstrate their validity through formal proofs. A first step towards such a framework is the precise and formal formulation of problems, allowing for methodical analysis. This is ensured through a set of rules that determine the validity of expressions, known as *syntax*. A language is then a set of expressions generated freely from these rules. For instance, we can consider a language that includes a as a *propositional symbol*, representing a statement, as well as the composition $b \vee c$ for any propositional symbols b and c. Thus, $a \vee a$ would be a syntactically well-formed statement. *Semantics* deals with the interpretation of such expressions within a given context. According to the usual interpretation of $\vee$ as the logical 'or', semantics would define that $b \vee c$ is true if and only if either b, c, or both are true. The ways in which we can infer new statements from given ones are defined by *inference rules*, sets of which form *calculi*. This falls within the realm of *proof theory*. Calculi need to possess two properties: First, their rules should be sufficient to prove any true statement that can be formulated in the system (called *completeness*), and more importantly, we should only be able to prove true statements (called *soundness*) [57]. Whether a calculus possesses these properties can be established through meta-theoretical results.

A logic is a system composed of these three components: syntax, semantics, and proof theory. *Propositional logic*, from which the exemplary definitions given above originate, is far from the only such system in use today. Countless logics have been introduced over the decades to formalize various disciplines. Modal and other non-classical logics [15] for instance permit reasoning about concepts like necessities, obligations, and knowledge, thus finding applications in fields like law and philosophy. Furthermore, logic and mathematics have been tightly linked ever since formal logic and reasoning have found their way into the formulation of mathematics in the 19th century [43], which consequently played a crucial part in advancing formal logic. This has led, among other things, to the introduction of type systems [73], which are necessary to eliminate paradoxes (like the infamous Russell's paradox) by requiring each object to be assigned a *type*, thus avoiding self-references. Nowadays, computer sciences have become an essential asset in the field and have given rise to *reasoning systems* that can either support human reasoning as *proof assistants* or fully autonomously prove a conjecture from given assumptions. The latter kind of systems, known as *automated theorem provers* (ATPs) [57], are based on a logic and implement the inference rules of a calculus to derive proofs for a given *conjecture* based on assumptions, commonly referred to as *axioms*. While such systems find wide applications in fields like software and hardware verification (e.g., ProVerif in security protocol verification [16]), mathematics (e.g., the ATP EQP proved the open Robbins Problem [51]), and philosophy (e.g., to prove statements in analytical philosophy [46]), they are often not used to their full potential for two reasons: Firstly, the soundness of the underlying calculus of a system does not automatically transfer to the ATP, as errors in implementation or faulty representation of inference rules can occur. In applications where the correctness of every proof must be guaranteed, as in hardware verification or in proof assistants, fully automated provers can thus not be readily used. Secondly, cooperation among specialized provers could theoretically enable them to derive proofs beyond the capabilities of individual systems. However, incompatible output encodings of individual systems make this difficult to achieve in practice. Dedukti [6] is a logical framework that allows the expression of different object-logics, aiming to overcome these hurdles by providing means to express, combine, and check proofs originating from different systems. A number of systems and proof libraries have already become part of the Dedukti network, including some ATPs that can directly output verifiable proofs. Until now, only systems capable of *first-order logic* (FOL) reasoning, which is very commonly used but limited in its expressive power, have been made available. This thesis lays theoretical foundations to extend Dedukti-based verification to *higher-order logic* (HOL).

1.1 Automated Reasoning and the Leo-III System

The general automation approach followed by HOL ATPs is illustrated in Fig. 1. Problems are first expressed in a machine-readable textual syntax, as shown by the 'Input Problem' in Fig. 1. The *TPTP*

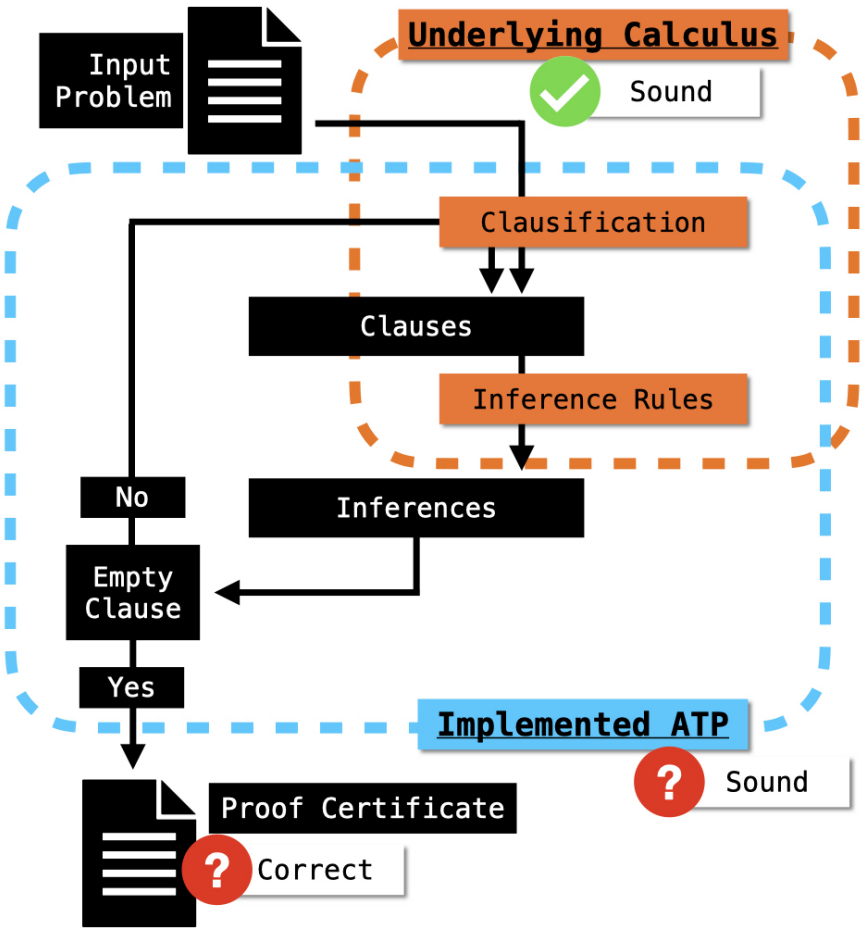

Figure 1: HOL ATP workflow. Black fields are the entities processed by the ATP and parts of the diagram enclosed by the area bordered with a light blue dotted line represent the internal reasoning process of the ATP system. The steps implementing the underlying calculus are highlighted in orange.

framework [69] provides a unified syntax for expressing the logics commonly used by theorem provers and standardizes the formulation of reasoning problems into sets of formulae categorized by roles (e.g., axiom or conjecture). The TPTP syntax has become the de facto community standard for ATP input problems. Reasoning in state-of-the-art ATPs follows an approach called *refutation*, which does not try to directly prove the conjecture (for instance, by constructing it from the assumptions) but instead demonstrates that the negation of the conjecture gives rise to contradictions with the axioms and must thus be false. Following this principle, the conjecture is therefore negated and transformed into an easier to manipulate normal form along with the assumptions. To this end, the formulae are transformed into a conjunction (statements connected by $\wedge$, the logical 'and') of *clauses*, which are themselves disjunctions (statements connected by $\vee$, the logical 'or') of *literals*, the basic units of such formulae. This *clausification* is the first operation highlighted in orange in Fig. 1 and invokes a step called *skolemization* in the presence of *existential quantifiers* (logical operators that allow the formulation of 'there exists' statements, denoted as $\exists$). The process removes the quantifiers through the introduction of *Skolem terms*, functions mapping the variables that are existentially quantified to a specific value satisfying the existential statement [57]. By definition, a clause is true if any of its literals are true (note that an empty clause can therefore never

be true), while the conjunction of all clauses is true only if none of the clauses is false. ATPs thus show that the conjunction representing the transformed problem with the negated clause is false by deriving an empty clause through the continuous application of inference rules. After each such application step (signified by the 'Inference Rules' highlighted in orange in Fig. 1), the system checks whether the empty clause is among the newly derived formulae. If this is not the case, the new formulae are clausified and added to the set of clauses and the process begins anew. Since the problem of proving a conjecture is undecidable in general, this process can run indefinitely and thus has to be terminated if the empty clause is not found within a given time limit. If the proof search is successful, a proof is reconstructed by selecting the inferences that led to the empty clause and given in the output as a step-by-step derivation [57], this is the 'Proof Certificate' in Fig. 1. The TPTP also offers a standardized syntax for these outputs, the TSTP [67], which not only lists the derived formulae but also provides information about the parent formulae and the applied rules.

What kinds of problems a specific prover can handle heavily depends on the logic implemented by concrete systems. FOL builds on the connectives of propositional logic and extends them with 'for all' statements over concrete objects or variables, using *universal quantifiers* (denoted as $\forall$). The resulting system can be used to phrase problems in many different application fields. While this makes it a common choice for ATPs, prominent examples of such systems being E [61] and Vampire [49], there also exist specialized systems for non-classical logics, for instance, MleanCoP [55] for the automation of first-order modal logics. Alternatively, ATPs can be based on higher-order logic (HOL) [14], which introduces a simple yet expressive syntax based on the simply typed λ-calculus [25]. Furthermore, HOL removes the restriction of quantification to individuals of the logic, as found in FOL. This, in fact, makes HOL versatile enough to encode the semantics of many non-classical logics, among them the previously mentioned modal logics, which can thus be embedded in HOL and problems can be solved using HOL reasoning [64]. This expressiveness, however, comes at the expense of additional challenges in automation [57].

Leo-III [62, 63] is an ATP that fully leverages the expressiveness of the higher-order logic it is based on (*Extensional Type Theory* (ExTT) [13]) by providing automated translations of various non-classical logics to HOL through a built-in shallow embedding. The collection of the non-classical logics Leo-III can handle this way is steadily growing in co-development with the TPTP syntax and in some cases offers even more flexibility than specialized systems [71]. Furthermore, Leo-III has also been extended to handle *Rank-1 polymorphism* [65], which represents a restricted version of the $\lambda 2$ system [8]. This enriches the type system by allowing a restricted use of quantification and HOL term operations within types, which adds further expressive capabilities.

1.2 Verification

Although standard ATPs cannot guarantee error-free proofs, established provers have gained the community's trust by consistently performing reliably and for instance producing the expected results over large problem sets in system competitions like the annual CASC [68]. In application areas where such empirical verification is not sufficient, formal verification comes into play. A very involved way of ensuring the correctness of proofs is the verification of the implementation of provers themselves. A more feasible alternative is the verification of individual proofs [66]. To this end, a trace containing details about all of the steps taken in the proofs produced by ATPs can be used to check the inferences and thus verify their correctness.

A platform that has made it its goal to provide a common framework to express and ensure the trustworthiness of proofs originating from different logics and systems is the *Dedukti framework* [6]. Dedukti implements the $\lambda\Pi$-calculus modulo theory, combining two powerful concepts: Firstly, the $\lambda\Pi$-*calculus*, or *Edinburgh logical framework*, extends HOL with dependent types. This makes it possible to construct types parameterized by terms. Secondly, *modulo theory* adds *rewrite rules* that can be used to define term or type symbols [33, 34] by replacing any of their occurrences. The $\lambda\Pi$-calculus provides a meta-logic capable of expressing the object-logics, axioms, deduction rules, and proofs of various theorem provers [28]. Moreover, the *Curry-Howard correspondence* [72] and the *Brouwer–Heyting–Kolmogorov interpretation* [53], respectively representing the ideas that propositions can be interpreted as types of

their proofs and logical connectives as type constructors, can be realized through the definition of rewrite rules. This allows propositions to be embedded as types, reducing proof checking to type checking, which is decidable given a properly defined rewrite system [6]. Dedukti not only provides a theoretical framework and a machine-readable syntax to formulate such encodings, but also offers type checkers for Dedukti files. Since the correctness of typing in the encoding corresponds to the correctness of proofs in the original system, these type checkers, in effect, verify the proofs.

Lambdapi[1] is an interactive proof assistant and type checker that can not only read and output Dedukti files, but also offers a more user-friendly syntax than Dedukti. Furthermore, it provides a number of additional features, such as interactive proof-scripts, in which commands (called *tactics*) can facilitate proofs. These innovations make the automated encoding of proofs accessible to systems like Leo-III, which have a more complex and diverse calculus.

Ever since the introduction of Dedukti, there has been an ongoing effort in the community to express proofs of different systems as well as existing proof libraries within the framework and modify provers to directly output proofs in Dedukti or Lambdapi syntax. There are three systems realizing the latter: ArchSAT[2] [23], a theorem prover integrating approaches of first-order reasoning in SMT-solving, and the FOL ATPs ZenonModulo [31] and iProverModulo [21]. These can then be used in a second approach, which involves the reconstruction of the individual steps of proofs written by systems with other output formats. This way, many proof snippets in the Dedukti format can be obtained and recombined to form a complete Dedukti proof. This is implemented by the tool EKSTRAKTO to verify TSTP proofs of CNF problems [40]. There also exists a version of the tool GDV [66], which relies on ZenonModulo to prove the semantic relationships between parent and child formulae of inferences in TSTP proofs. Lastly, tools that translate proof libraries of renown proof assistants to Dedukti have already been made available, examples being Coqine [20] for Coq, Krajono[3] for Matita, Isabelle_Dedukti[4] for Isabelle and Holide [5] for OpenTheory. Further such software is under development, one example being hol2dk for HOL-Light[5].

1.3 Thesis Scope and Outline

One system currently missing from this list is an HOL ATP directly outputting proofs verifiable in the Dedukti framework. Closing this gap by enabling an existing prover with an option to output such proofs can, however, be challenging since, in provers where the ease of a Dedukti encoding was not a concern during development, there are typically a larger number of inference rules that can be harder to encode. Furthermore, transformations like the permutation of the literals of a clause can occur as a result of the implementation rather than a deliberate application of an inference rule. While this is unproblematic in the original prover, it adds numerous additional factors that have to be accounted for in a Dedukti encoding and hence makes outputting fully verifiable proofs significantly harder. This thesis aims at developing strategies to address these challenges and establish a framework for encoding HOL proofs, both generally and specifically for Leo-III.

A number of restrictions are necessary within the scope of this thesis: First, clausification steps will be excluded as they involve skolemization. While this process preserves satisfiability and unsatisfiability, which is sufficient for deriving the empty clause from the negated conjecture and thus showing provability, it does not maintain equivalence [57]. Since this makes verification more challenging, developing a suitable verification strategy or adapting existing methods and software used in FOL verification, is left for future work. Furthermore, Leo-III employs polymorphic logic, but the verification and encoding here is restricted to the monomorphic version of the logic and calculus rules. However, the work is designed to be extendable to polymorphic HOL in the future. Lastly, only a subset of the calculus rules was considered, as encoding the full calculus was not feasible within the scope of this project. The chosen rules are representative of different kinds of inferences, providing insight into the diverse challenges in encoding HOL automated reasoning.

[1] https://github.com/Deducteam/lambdapi
[2] https://github.com/Gbury/archsat
[3] https://deducteam.gitlabpages.inria.fr/krajono/
[4] https://github.com/Deducteam/isabelle_dedukti
[5] https://github.com/Deducteam/hol2dk

The outline of both the project presented here and the thesis itself is visualized in Fig. 2.

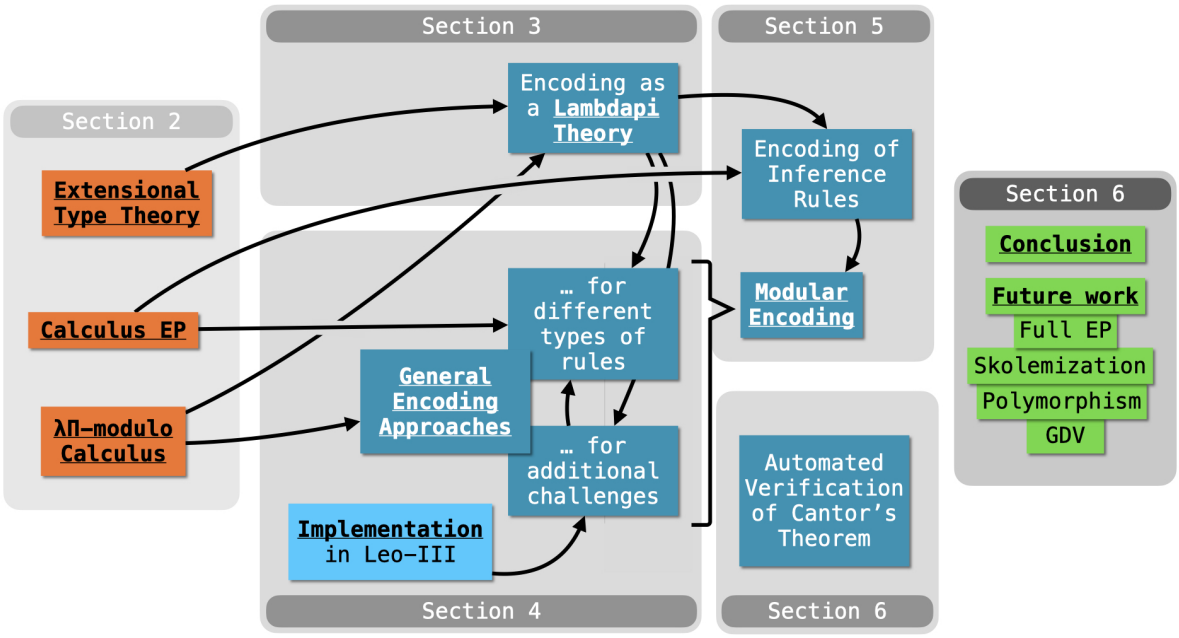

Figure 2: Contextual Overview of Discussed Topics: Underlying theories are marked in orange, the implementation of the Leo-III system is light blue, turquoise fields signify encodings, and green fields for the conclusion and future work. Arrows indicate dependencies of the topics. The gray backdrops divide the topics into the sections of this thesis.

Some further background information on ExTT and the calculus Leo-III is based on (EP), as well as the logical framework implemented by Dedukti (λΠ-modulo Theory), will be introduced in Sect. 2. The basis of the encoding is then provided in Sect. 3 through the definition of a so-called Lambdapi theory that represents the object-logic (ExTT) within Lambdapi. The various demands and challenges posed by the encoding of specific rules and their implementations in an ATP are categorized in Sect. 4, and encoding approaches are developed. Based on these approaches and the encodings of the individual calculus rules, a modular encoding for steps in Leo-III proofs is derived in Sect. 5. The suitability of the derived encoding for automation will be empirically verified through a partial implementation aimed at fully automatically providing a verifiable proof of Cantor's theorem, this is presented in Sect. 6.3. Sect. 7 concludes the thesis with a summary and a discussion of future work.

2 Preliminaries

In the following section, the theoretical foundations of the thesis is laid out. This includes introductions to HOL, the EP calculus used by Leo-III, and the λΠ-calculus modulo theory implemented by Dedukti. Unless otherwise noted, the definitions provided here closely follow the publications introducing Leo-III [62] and the λΠ-calculus modulo [28]. Notation is adopted in most parts but adjusted to ensure uniformity.

2.1 Higher-Order Logic

The version of HOL originally proposed by Church [25] included a number of axioms, some of which are not commonly used in automated theorem proving today. Instead, Leo-III and many other higher-order ATPs implement ExTT [13] that originates from the works of Henkin [42], Andrews [2], and others. Here, some assumptions, like the axiom of infinity, are omitted while retaining notions of extensionality. In the following, ExTT and HOL are used interchangeably.

2.1.1 Syntax

Types. HOL is a typed logic, meaning that each term is categorized through the assignment of a type. The base types, denoted as o and ι, represent the types of truth values and individuals, respectively. Apart from these, further *primitive types* can be introduced if necessary. The simple types then include the primitive types as well as the construction $T \to V$ for any simple types T and V. Such a construct is called a *function type* and expresses that the typed term maps an argument of type T to a term of type V. The notation here is right-associative. The set of all simple types is denoted as $\mathcal{T}$. Types are indicated as subscripts (for instance a_o for the propositional symbol a) or are omitted when clear from the context. A type declaration can also be given explicitly in the style $a : o$.

Terms. The atomic entities of HOL terms are *constants* and *variables*. The former are typed symbols that, depending on the type, can represent functions, predicates, and operators. $\mathcal{V}$ denotes the set of all typed variable symbols, with the number of symbols of each type being (countably) infinite. Similarly, the *signature*, denoted Σ, is the set of all typed symbols, excluding variables. The syntax of HOL is then given by...

$$s, t \coloneqq c_T \in \Sigma \,|\, \boldsymbol{x}_T \in \mathcal{V} \,|\, (\lambda x_T.s_V)_{T\to V} \,|\, (s_{T\to V}t_T)_V$$

The constants (denoted here by lowercase letters as in c_T) and variables (denoted here by calligraphic lowercase letters as in $\boldsymbol{x}_T$) are used to form *anonymous functions* through so-called *abstractions* written $\lambda \boldsymbol{x}.s$. Occurrences of $\boldsymbol{x}$ in the term s of such a construction are called *bound*, variables not bound are called *free*, and the set of free variables of a term t is denoted as $fv(t)$. A term without free variables is called *ground*. An unnamed function mapping a propositional variable $\boldsymbol{x}_o$ to the term $\boldsymbol{x} \vee \boldsymbol{x}$ would thus for instance be written $\lambda \boldsymbol{x}.\boldsymbol{x} \vee \boldsymbol{x}$ and $\boldsymbol{x}$ would be a bound variable. The *application* of arguments in functions is indicated by juxtaposition and replaces the variables bound by λ-abstraction of the anonymous functions with the applied terms. More than one argument can be applied in this fashion through subsequent application of individual arguments, called *currying*, if the function type permits it. In such cases, the application is by convention left-associative and an abbreviated notation for nested application $s, t^1, ..., t^n$ is $s, \overline{t^n}$. The evaluation of a function application is done by so-called *β-reduction*, denoted as $\to_\beta$. Evaluating the previously introduced anonymous function at the argument a_o is thus written $(\lambda \boldsymbol{x}.\boldsymbol{x} \vee \boldsymbol{x})a \to_\beta a \vee a$. β-expansion performs the complementary operation, and β-conversion is the transitive closure by context of β-reduction and β-expansion. η-expansion allows to extend terms s to constructions of the form $\lambda \boldsymbol{x}.s\boldsymbol{x}$, where $\boldsymbol{x}$ does not occur free in s, η-reduction again allows the opposing operation. α-conversion renames every bound occurrence of variables consistently. Two terms that only differ in the naming of their bound variables can thus be identified after α-conversion and are called *syntactically equivalent*, denoted as $s_T \equiv t_T$. In such cases, α-conversion will be treated implicitly, and terms will be regarded as equivalent. For the conversions $\star = \{\alpha, \beta, \eta\}$, $\to_\star^*$ denotes the transitive and

reflexive closures by reduction, and two terms fulfilling $s \leftrightarrow_\star^* t$ can be identified modulo the respective conversions and are thus equivalent with respect to them, denoted $s \equiv_\star t$. As a result of the introduction of types, β-reduction is confluent and terminating in the case of HOL [9].

A substitution σ replaces variables of a term t it is applied to, written $t\sigma$, with terms. It is defined as a mapping of variables x^i to terms s^i that is given by $\sigma \equiv s^1/x^1, , ..., s^n/x^n$.

Connectives. It is possible to choose different connectives as the primitive one(s) and define the others with regard to them. In the case of Leo-III, the equality predicate, written $=^T$ for terms of type T, is chosen, and $\top_o$, $\bot_o$, $\neg_{o\to o}$, $\wedge_{o\to o\to o}$, $\Leftrightarrow_{o\to o\to o}$, and $\Pi^T_{(T\to o)\to o}$ (with $\forall x T.t_o$ as a shorthand notation) are defined with respect to it (see Fig. 1 in [62] for the precise definitions). Based on them, the remaining connectives ($\Rightarrow_{o\to o\to o}$, $\vee_{o\to o\to o}$, and $\exists_{(T\to o)\to o}$) can then be defined as usual. The choice of equality as the primitive connective is beneficial in proofs of the meta-properties of the calculus, but in the implementation of Leo-III, connectives are treated as primitive for the sake of efficiency.

2.1.2 Semantics

The underlying principles of HOL semantics are recalled here, the formal definitions and a more thorough discussion can be found in the literature [3].

We assume a *universe* U_0, which is a collection of non-empty sets with standard properties. Each type corresponds to a set within U_0 that includes the specific values or entities a type can represent, known as *denotations*. For a type $T \in \mathcal{T}$, its set of denotations is referred to as $\mathcal{D}_T \in U_0$. A collection $(\mathcal{D}_T)_{T\in\mathcal{T}}$ is called a *HOL frame* if $\mathcal{D}_\iota \neq \emptyset$, $\mathcal{D}_o := \{T, F\}$, and $\mathcal{D}_{V\to T} \subseteq \mathcal{D}_V^{\mathcal{D}_T}$. In a standard frame $\mathcal{D}_{T\to V}$ includes the full set of functions from $\mathcal{D}_T$ to $\mathcal{D}_V$.

Denotations of concrete objects can then be defined with respect to a signature Σ and a standard frame $\mathcal{D}$ by an *interpretation* $\mathcal{I}$, which maps each constant symbol $c_T \in \Sigma$ to an element in $\mathcal{D}_T$. Similarly, a *variable assignment* g maps any variable to a denotation with the corresponding type. A frame and an interpretation form a *model* $\mathcal{M} \equiv (\mathcal{D}, \mathcal{I})$. The *value* of a term can then be obtained by the *valuation* function v, that maps terms to their denotations with regard to a model $\mathcal{M}$ and a variable assignment g.

If standard frames are used, the resulting framework is referred to as *standard semantics*, which, as a result of Gödel's incompleteness theorem [39], are incomplete. This issue was addressed by Henkin [42] and Andrews [1], who introduced so-called *general models*, that restrict the domain of function types to a subset sufficient to provide denotations for all syntactic objects. The use of such general models yields HOL with *general semantics*, for which completeness can be shown. Therefore, we will assume HOL with general semantics in the following.

A term s_o is *valid* with regard to a general model $\mathcal{M}$, written $\mathcal{M} \vDash s_o$, if its valuation with respect to all variable assignments g, written $v^{\mathcal{M},g}$, evaluates to true. If this is the case for all $\mathcal{M}$, s_o is *valid*. We call t_o a *consequence* of s_o, written $s_o \vDash t_o$, if $\mathcal{M} \vDash t_o$ holds for any $\mathcal{M}$ such that $\mathcal{M} \vDash s_o$. Analogously, if this relationship holds for all $t_o^i \in \Delta$, then $\Delta \vDash s_o$.

2.2 The Calculus EP

2.2.1 Notation

Literals. Literals are generally encoded using tuples of terms s and t called *equations*, denoted as $s \simeq t$. Literals are *signed* equations, written as $l \equiv [s \simeq t]^\alpha$, where $\alpha \in \{tt, ff\}$ is the truth value of the literal. $\overline{\alpha}$ then denotes the reversed truth value, i.e. $\overline{tt} \equiv ff$ and $\overline{ff} \equiv tt$. In clausification, literals encoding equality with the usual $=$ can be lifted to equational literals. Differentiating between equations and equalities however makes it possible to define precisely on what structures the inference rules operate. Non-equational literals, denoted $[s]^\alpha$, can be transferred to this notation through a presentation as an equality with $\top$, where, by convention, $\top$ is written on the right-hand side (as in $[s \simeq \top]^\alpha$). Notably, there is a correspondence between $[s \simeq \top]^\alpha$ and $[s \simeq \bot]^{\overline{\alpha}}$. An *atomic* literal is either equational, or has a head symbol that is not $\beta\eta$-equivalent to any logical connective.

The validity of a literal l, written $\mathscr{M}, \mathscr{g} \vDash l$, is given with respect to a model $\mathscr{M}$ and a variable assignment $\mathscr{g}$ if either $\mathscr{M}, \mathscr{g} \vDash s = t$ and $\alpha \equiv tt$ or conversely $\mathscr{M}, \mathscr{g} \vDash s \neq t$ and $\alpha \equiv ff$. Satisfiability is given if there exists a $\mathscr{g}$ such that $\mathscr{M}, \mathscr{g} \vDash l$ and validity with respect to $\mathscr{M}$, denoted $\mathscr{M} \vDash l$, is given if $\mathscr{M}, \mathscr{g} \vDash l$ holds for any $\mathscr{g}$.

Clauses. Disjunctions of literals $(l_1 \vee ... \vee l_n)$ form clauses. Given two clauses $\mathscr{C}$ and $\mathscr{D}$, $\mathscr{C} \vee l$ denotes the clause containing all literals of $\mathscr{C}$ as well as the literal l and $\mathscr{C} \vee \mathscr{D}$ denotes the clause containing all literals of $\mathscr{C}$ and $\mathscr{D}$. Every clause can be transformed to a so-called *clause normal form* (CNF), which is given if all literals are atomic.

The validity of $\mathscr{C}$ holds with respect to a model $\mathscr{M}$ and a variable assignment $\mathscr{g}$, written $\mathscr{M}, \mathscr{g} \vDash \mathscr{C}$ if any literal l of $\mathscr{C}$ is valid. Satisfiability as well as validity of $\mathscr{C}$ with respect to $\mathscr{M}$ are then defined analogously to their corresponding definitions in the case of literals. Note that neither the empty clause, nor a clause only containing literals of the form $[x\,\overline{s^i} \simeq y\,\overline{t^i}]^{ff}$, are valid in any HOL model $\mathscr{M}$ [42].

Substitutions. Substitutions for literals $(l\sigma)$ and clauses $(\mathscr{C}\sigma)$ are each defined component-wise. Let π denote a position. Then $s|_\pi$ and $s[t]_\pi$ respectively denote the subterm of s at positions π and the term resulting from the replacement of such a subterm at position π by a term t.

Calculus rules. An inference rule has the form

$$\frac{\mathscr{C}_1 \qquad ... \qquad \mathscr{C}_n}{\mathscr{C}'}\,(name)$$

and postulates that the conclusion $\mathscr{C}'$ can be drawn based on the (variable disjoint) assumptions $\mathscr{C}_1$, ..., $\mathscr{C}_n$, where $\mathscr{C}'$ and all $\mathscr{C}_i$ are clauses. A set of inference rules forms a calculus.

Given such a calculus $\mathscr{R}$ and a set of clauses $\mathfrak{C}$, the clause $\mathscr{C}$ is considered derivable from $\mathfrak{C}$ by $\mathscr{R}$, denoted $\mathfrak{C} \vdash_{\mathscr{R}} \mathscr{C}$, if there exists a sequence of clause sets $\mathscr{C}_0\mathscr{C}_1 \ldots \mathscr{C}_n$ such that: $I)$ $\mathscr{C}_0 \equiv \mathscr{C}$, $II)$ $\mathscr{C} \in \mathscr{C}_n$ and $III)$ For each $0 < i < n$, there is a rule $r \in \mathscr{R}$ such that $\mathscr{C}_{i+1}$ is equivalent to $\mathscr{C}_i \cup \mathscr{D}$, where $\mathscr{D}$ denotes the conclusion of the application of r and clauses $\mathscr{C}_i$ denote the premises.

2.2.2 Extensional Higher-Order Paramodulation

As discussed previously, automated theorem proving generally proves conjectures through refutation. In this process, the conjecture is negated and transformed to CNF along with the given premises. The inference rules of a chosen calculus are then repeatedly applied to the set of clauses in an attempt to derive $\bot$ (denoting a contradiction). The choice of a calculus well-suited for automation is therefore an essential step in the design of an efficient system.

Classical calculi used in FOL automated theorem proving are commonly based on resolution, a rule that concludes $[l_1 \vee l_3]^{tt}$ from two clauses $[l_1 \vee l_2]^{tt}$ and $[l_3 \vee \neg l_2]^{tt}$ [57]. As detailed in [62, 63], HOL automated reasoning still builds upon some of the principles of such FOL approaches, but several areas require additional techniques and adaptations, since a direct adoption of FOL strategies would be insufficient: In the presence of quantifiers, some terms may need to be instantiated appropriately to be compatible with the inference rules. *Unification* is the process of finding a *unifier*, a substitution that binds variables of both terms and renders them equivalent [57]. Determining that two terms are not unifiable is also informative in automated reasoning, because it limits the possible inferences between clauses that have to be considered. The successful application of unification in FOL ATP systems [60] is therefore considered one of the milestones in automated reasoning. While unification can be implemented as an efficient subroutine in FOL, the undecidability of unification in HOL presents a challenge. Another hurdle is that HOL inference rules taking clauses in CNF as premises can result in conclusions that are not in normal form, thus a single clausification step at the very beginning of the reasoning process is not sufficient. Furthermore, the handling of equational reasoning in classical resolution-based approaches without rules dedicated to this purpose is insufficient.

EP, a calculus for HOL paramodulation in ExTT, as proposed in [10, 37], combines and further refines different techniques to address these problems through the additon of dedicated calculus rules. Unification is handled through the introduction of so-called *unification constraints*, which take the form of negative equality literals as first suggested in [48] and refined in [37]. Intuitively, unification constraints represent an equality that has to be satisfied in order for the conclusion of an applied calculus rule to hold: Let, for instance, $\mathscr{C} \vee [s \simeq t]^{ff}$ be a clause with the unification constraint $[s \simeq t]^{ff}$. If $s \simeq t$ can be shown, the literal $[s \simeq t]^{ff}$ is false and hence $\mathscr{C}$ has to be valid in order for the whole clause to be valid. Rather than attempting to unify terms before the application of inference rules, such constraints represent the condition under which the inference rules derive their conclusions and are added to the clauses representing the conjecture of the rules. A number of unification rules are then used to process such constraints. A further set of inference rules is introduced for clausification, making it possible to transform freshly generated clauses to CNF in between the application of other inference rules if necessary. The handling of equality on a calculus level is realized through *paramodulation* rules [59], that are based on the idea that terms known to be equal can replace each other. This results in rules that, for instance, deduce $[p_{T\to o}\, s]^{\alpha}$ from both $[p_{T\to o}\, t]^{\alpha}$ and $[s \simeq t]^{tt}$.

The calculus was later modified to handle extensionality, a concept necessary in the context of HOL, through the addition of inference rules [12, 13]. Leo-III [62, 63] not only implements this calculus but also complements the core calculus with a number of additional rules helpful in automation.

2.2.3 Core-Calculus

The rules of the calculus are taken over from [62], where the soundness of the core calculus is also shown. Note that the core calculus also contains clausification rules, they are however not recalled in the following due to the restriction to formulas already in CNF in this project.

Primary Inference Rules. The primary inference rules of the calculus EP are displayed in Fig. 3.

Paramodulation

$$\frac{C \vee [s_T \simeq t_T]^{\alpha} \qquad D \vee [l_V \simeq r_V]^{tt}}{[s[r]_{\pi} \simeq t]^{\alpha} \vee C \vee D \vee [s|_{\pi} \simeq l]^{ff}}\ (\text{Para})^{\dagger}$$

Equal Factoring

$$\frac{C \vee [s_T \simeq t_T]^{\alpha} \vee [u_T \simeq v_T]^{\alpha}}{C \vee [s_T \simeq t_T]^{\alpha} \vee [s_T \simeq u_T]^{ff} \vee [t_T \simeq v_T]^{ff}}\ (\text{EqFac})$$

Primitive Substitution

$$\frac{C \vee [\hbar_T \overline{s^i_{T_i}}]^{\alpha} \qquad g \in \mathcal{GB}_T^{\{\neg,\vee\}\cup\{\Pi^V,=^V \mid V\in\mathcal{T}\}}}{C \vee [\hbar_T \overline{s^i_{T_i}}]^{\alpha} \vee [\hbar \simeq g]^{ff}}\ (\text{PS})$$

†: $s|_{\pi}$ is of type V and $fv(s|_{\pi}) \subseteq fv(s)$

Figure 3: Primary Inference Rules of EP

As discussed previously, paramodulation (Para) provides the means to replace equals with equals. For this purpose, a positive equality literal $[l_V \simeq r_V]^{tt}$ originating from one clause is used to rewrite a subterm $s|\pi$ of literals in another clause. The conditions that need to be fulfilled for this are given by the unification constraint encoding the unifiability of the left-hand side of the equality literal and the respective subterm $[s|_\pi \simeq l]^{ff}$, as well as the remaining literals D of the clause containing the literal used for rewriting.

Factoring is a process in automated theorem proving that replaces two literals in a clause with their *factor*, which is the result of applying their most general unifier (MGU) [57]. *Equal factoring* in Leo-III implements this through the use of unification constraints. Given a clause containing two literals $[s_T \simeq t_T]^\alpha$ and $[u_T \simeq v_T]^\alpha$, the unification constraints $[s_T \simeq u_T]^{ff}$ and $[t_T \simeq v_T]^{ff}$ encode the unifiability of both their left- and right-hand sides. If a substitution fulfilling both of these conditions can be found, the original literals are unifiable. Consequently, in the conclusion of the rule (EqFact), only the first literal $[s_T \simeq t_T]^\alpha$ remains.

The logical structure of non-equational literal with a variable $\hbar_{\mathcal{T}}$ as the head symbol heavily depends on said variable head. *Primitive substitution* (PrimSubst) uses *general bindings*, denoted $\mathcal{GN}$, to stepwise construct more complex terms that can instantiate the head symbol using abstractions and logical connectives. The substitution of the variable with a concrete general binding g is then encoded in a corresponding unification constraint $[\hbar \simeq g]^{ff}$ that is added to the clause.

Extensionality. The extensionality rules of Leo-III are provided in Fig. 4.

Propositional Extensionality

$$\frac{C \vee [s_o \simeq t_o]^{tt}}{\begin{array}{l} C \vee [s_o]^{tt} \vee [t_o]^{ff} \\ C \vee [s_o]^{ff} \vee [t_o]^{tt} \end{array}}\ \text{PBE} \qquad \frac{C \vee [s_o \simeq t_o]^{ff}}{\begin{array}{l} C \vee [s_o]^{tt} \vee [t_o]^{tt} \\ C \vee [s_o]^{ff} \vee [t_o]^{ff} \end{array}}\ \text{NBE}$$

Functional Extensionality

$$\frac{C \vee [s_{T\to V} \simeq t_{T\to V}]^{tt}}{C \vee [s\ X_T \simeq t\ X_T]^{tt}}\ \text{PFE}^\dagger \qquad \frac{C \vee [s_{T\to V} \simeq t_{T\to V}]^{ff}}{C \vee [s\ sk_T \simeq t\ sk_T]^{ff}}\ \text{NFE}^\ddagger$$

†: where X_T is fresh for C,
‡: where sk_T is a Skolem term

Figure 4: Extensionality Rules of the Calculus EP

Leo-III employs HOL with equality as a primitive connective. Complete reasoning requires relating equality to equivalence for terms of type o and defining criteria for the equality of function terms when arguments are applied. In HOL, this is articulated through *propositional extensionality* (propExt) and *functional extensionality* (funExt), which can be defined as the following axioms [11]:

$$\text{propExt} := \forall p_o.\forall q_o.(p \Leftrightarrow q) \Rightarrow p =^o q$$
$$\text{funExt} := \forall f_{T\to V}.\forall g_{T\to V}.(\forall x_T.f\,x =^V g\,x) \Rightarrow f =^{T\to V} g$$

Note that the right-to-left direction of both principles is trivially true by the usual equality properties. A representation of these principles is crucial for completeness in ExTT. However, merely adding such axioms to the search space would necessitate automated provers to guess useful instantiations of propExt

and funExt in order for the principles to be used [12]. This situation is analogous to the addition of the *cut rule* in automated reasoning, which postulates that a conclusion A of one subproof used as a hypothesis in another subproof represents an intermediate step that can be cut from the proof [58]. Although this simplifies proofs, it also requires provers to guess the instantiation of A. Since this process can be time-consuming and may not result in a useful instantiation at all, calculi for automated provers are typically designed to be cut-free. [12] relates the introduction of axioms for extensionality to the well-known issues of the cut rule. It has thus been proposed to avoid such *cut simulation* by representing extensionality principles through the addition of calculus rules rather than axioms to achieve calculi that are both efficient and complete [12, 13]. This is implemented in Leo-III in the extensionality rules (Fig. 4).

(PBE) represents propositional extensionality for positive literals and allows the deduction of the mutual implication of s_o and t_o by forming two new clauses, each containing the implication in one of two directions. (NBE) handles the inequality of s_o and t_o by deducing that either s_o or t_o is correct while the the other is false. Functional extensionality for positive literals is expressed by (PFE), which states that the equality of two functions still holds if the same argument is applied to both. Conversely, (NFE) addresses negative literals, postulating the existence of concrete arguments on which the function terms differ if they are not equal. In Leo-III, this is implemented through the introduction of a Skolem term.

Unification. As we have seen, the problem of undecidability of unification in HOL is addressed by the addition of unification constraints to encode the conditions under which inferences are valid. The unification rules in Fig. 5 are a variation of Huet's unification procedure [45], integrated into the calculus to enable an effective subroutine for unification that can be applied after the application of inference rules generating unification constraints [62, 63].

Unification

$$\frac{C \vee [s_T \simeq s_T]^{ff}}{C}\ (\text{Triv}) \qquad \frac{C \vee [\boldsymbol{x}_T \simeq s_T]^{ff}}{C\{s/\boldsymbol{x}\}}\ (\text{Bind})^{\dagger}$$

$$\frac{C \vee [c\,\overline{s_i} \simeq c\,\overline{t_i}]^{ff}}{C \vee [s_1 \simeq t_1]^{ff} \vee \ldots \vee [s_n \simeq t_n]^{ff}}\ (\text{Decomp})$$

$$\frac{C \vee [\boldsymbol{x}_{V\overline{U}}\,\overline{s^i} \simeq c_{V\overline{T}}\,\overline{t^j}]^{ff} \qquad g_{V\overline{U}} \in GB^{\{c\}}_{V\overline{U}}}{C \vee [\boldsymbol{x}_{V\overline{U}}\,\overline{s_i} \simeq c_{V\overline{T}}\,\overline{t_j}]^{ff} \vee [\boldsymbol{x} \simeq g]^{ff}}\ (\text{FlexRigid})$$

$$\frac{C \vee [\boldsymbol{x}_{V\overline{U}}\,\overline{s^i} \simeq \boldsymbol{y}_{V\overline{T}}\,\overline{t^j}]^{ff} \qquad g_{V\overline{U}} \in GB^{\{h\}}_{V\overline{U}}}{C \vee [\boldsymbol{x}_{V\overline{U}}\,\overline{s^i} \simeq \boldsymbol{y}_{VT}\,\overline{t^j}]^{ff} \vee [\boldsymbol{x} \simeq g]^{ff}}\ (\text{FlexFlex})^{\ddagger}$$

$\dagger$: where $\boldsymbol{x}_T \notin \text{fv}(s)$ $\quad$ $\ddagger$: where $h \in \Sigma$ is an appropriate constant

Figure 5: Unification Rules of the Calculus EP (Fig. 4 in [62])

While the rule (FlexFlex) is useful in proving the completeness of the calculus, it is not beneficial for the automation. Empirical evidence suggests the admissibility of the rule, it was hence omitted from the implementation of Leo-III and will also not be considered in the verification of Leo-III proofs. The rules (FlexRigid) and (Decomp) provide means to iteratively *solve* unification constraints, i.e., transform them until a unification constraint $[\boldsymbol{x}_T \simeq s_T]^{ff}$ is reached where $\boldsymbol{x}_T$ is not a free variable of s_T. Once this is achieved, the rule (Bind) can be applied to realize the condition encoded in the unification constraint by replacing any occurrence of $\boldsymbol{x}_T$ in the remaining clause C with s_T. The rule (Triv) simply derives

a simplified clause in cases where unification has transformed a literal to $[s_T \simeq s_T]^{ff}$, which is trivially false and can thus be omitted from the clause.

2.2.4 Extended Calculus

The extended calculus implemented in Leo-III involves a number of additional rules that are not part of the original calculus EP [10, 37]. These additions are primarily motivated by the efficiency of the automation. This includes rules that instantiate universally quantified variables according to heuristic criteria or replace instances of axiomatically defined equality or choice operators in the reasoning problem with the versions system-defined in Leo-III. Notably, many extensions aim at contracting clauses or literals or providing criteria under which clauses or literals can be omitted, thereby simplifying the set of clauses. Instead of reviewing the entire set of rules in the extended calculus, we will focus on a few of the most commonly applied and particularly relevant rules.

Formula Simplification. The well-known identities given in Fig. 6 are used to contract boolean expressions.

$s \vee s$	$\to s$	(Simp 1)	$s \wedge s$	$\to s$	(Simp 2)
$\neg s \vee s$	$\to \top$	(Simp 3)	$\neg s \wedge s$	$\to \bot$	(Simp 4)
$s \vee \top$	$\to \top$	(Simp 5)	$s \wedge \top$	$\to s$	(Simp 6)
$s \vee \bot$	$\to s$	(Simp 7)	$s \wedge \bot$	$\to \bot$	(Simp 8)
$t = t$	$\to \top$	(Simp 9)	$t \neq t$	$\to \bot$	(Simp 10)
$s = \top$	$\to s$	(Simp 11)	$s = \bot$	$\to \neg s$	(Simp 12)
$\forall \boldsymbol{x}_T.s$	$\to s$	(Simp 13)†	$\exists \boldsymbol{x}_T.s$	$\to s$	(Simp 14)†
$\neg \bot$	$\to \top$	(Simp 15)	$\neg \top$	$\to \bot$	(Simp 16)
			$\neg\neg s$	$\to s$	(Simp 17)

$\dagger$ if $x \notin \mathrm{fv}(s)$

Figure 6: Simplification Rules of Leo-III (Figure 2 in [62])

The rule (Simp) exhaustiveley applies the identities to the left- and right-hand sides of equality literals.

$$\frac{[l_1 \simeq r_1]^{\alpha_1} \vee \cdots \vee [l_n \simeq r_n]^{\alpha_n}}{[\mathrm{simp}(l_1) \simeq \mathrm{simp}(r_1)]^{\alpha_1} \vee \cdots \vee [\mathrm{simp}(l_n) \simeq \mathrm{simp}(r_n)]^{\alpha_n}}\ \text{Simp}$$

Rewriting.

$$\frac{C \vee [s \simeq t]^{\alpha} \qquad [l \simeq r]^{tt}}{C \vee [s[r\sigma]_p \simeq t]^{\alpha}}\ (\text{RW})^\dagger$$

$\dagger$: Where $s|_p \equiv l\sigma$ for some substitution σ and $l\sigma$ is bigger than $r\sigma$ with regard to a term ordering

Clauses consisting of only one single literal either encode equalities that must hold in the problem or, in the case of non-equational literals, identify boolean terms with $\top$ or $\bot$. (RW) thus utilizes such rewrite-clauses to rewrite other clauses in the problem and - if the former are unground - performs this operation modulo matching of the left-hand side of the rewrite-clause to substructures of other clauses using substitutions.

2.3 λΠ-Calculus Modulo Theory

2.3.1 λΠ-Calculus

The λΠ-calculus, also known as the Edinburgh Logical Framework [41], incorporates the dependent types originating from works of De Bruijn [30]. These are written as the product $\Pi x\!:\!T.\,S$ and allow a type

S to depend on a variable x of type T. Note that dependent types generalize function types: If x does not occur free in S, the dependent type $\Pi x\!:\!T.\,S$ is equivalent to $T \to S$. A common example for a dependent type is $\Pi x : Nat.array(x)$ [28] which parameterizes the family of types for arrays by their lengths. Instantiating this product with 2 yields the type of arrays of length two: $array(2)$. The extension of HOL with dependent types necessitates additional categories in the typing hierarchy. This becomes apparent when attempting to type *array*: In effect, *array* as used here must map a natural number to a type and is thus assigned the function type $Nat \to Type$, with the symbol $Type$ representing the type of types. This however raises the question what type can in turn be assigned to $Type$ itself or to constructions such as $Nat \to Type$. For this purpose, a new category called *Kind* is introduced. Hence, types can themselves be seen as terms of type $Type$, and terms can become parts of types if passed as an argument to dependent types, blurring the distinction between types and terms. This allows us to treat all instances as terms and define a unified syntax of the $\lambda\Pi$-calculus:

$$T, S = x \mid Type \mid Kind \mid \Pi x\!:\!T.\,S \mid \lambda x\!:\!T.\,S \mid T\,S$$

Typing rules are thus necessary to restrict the ways in which terms can relate to each other and ensure syntactically well-formed type assignments. They are given in figure 7.

Γ here denotes a context, which is a collection of declared type assignments and $\Gamma[t : S]$ denotes the extension of Γ with the type assignment $t : S$. A context is well-formed if it consists of well-formed type assignments. A *judgment* of the form $\Gamma \vdash t : S$ states that a term t has type S under the context Γ. The first three rules define the notion of well-formedness: *Empty* states that the empty context is well-formed and the rules $ObjectDeclaration$ and $TypeDeclaration$ generalize the concept of well-typedness to contexts extended with declarations of free variables as objects of a type A or as types (which themselves have type $Kind$). The following rules then define under what conditions certain judgments hold: $Type$ postulates that in any $well\text{-}formed$ context, $Type$ is of sort $Kind$ and $Variable$ that for any declaration $x : A$ in Γ, the judgment $\Gamma \vdash x : A$ holds. $Product1$, $Product2$, $Abstraction1$, $Abstraction2$ and *Application* govern over which terms can form products, abstractions and application and type the resulting constructions. The last two rules, $Conversion1$ and $Conversion2$, assert that the validity of the judgment $\Gamma \vdash t : B$ follows from the β-equivalence of two types or kinds A and B and a typing $t : A$ in a context Γ.

2.3.2 Rewriting

Rewrite rules are pairs (l, r) in β-normal form, denoted $l \to^{\Delta,A} r$, expressing that occurrences of l of type A can be replaced with r of type A in a context Δ. There is a distinct difference between deduction rules and rewrite rules, as well as the possible uses in automated reasoning systems: Deduction rules need to be applied explicitly, which involves non-deterministic search in automated systems. In contrast, rewrite rules should always be applied to any occurrence of the term on their left-hand side, making their automation a straightforward computation. The approach of replacing deduction rules by rewrite rules is explored in Deduction Modulo [33, 34]. Computation is introduced through rewriting and can be executed directly. Reasoning can then be done modulo rewriting by identifying terms according to the congruence introduced by the rewrite rules. This simplifies proofs by reducing the number of necessary deductive steps. Such rewrite rules are also added to the $\lambda\Pi$-calculus and introduce another reduction in addition to β-reduction ($t \to_\beta s$). Combining both reductions results in a relation $\to_{\beta\mathcal{R}}$ (where $\mathcal{R}$ is the set of rewrite rules) that applies if s can be β-reduced or rewritten to t. This is incorporated into the typing rules for the $\lambda\Pi$-Calculus by replacing $\equiv_\beta$ with $\equiv_{\beta\mathcal{R}}$ (the reflexive-symmetric-transitive closure of $\to_{\beta\mathcal{R}}$ by context) in $Conversion1$ and $Conversion2$, leading to the identification of terms after both rewriting and β-reduction in the conversion rules. Well-typedness of rewrite rules in a context $\Gamma\Delta$ then holds if $\Gamma\Delta$ is well-formed and the rewrite rule preserves typing, i.e. if both l and r share a type A.

The process of encoding problems and proofs in the resutling framework is sketched out in detail in section 3, but the underlying idea is the following: Propositions are encoded as types and assigned to terms, which then represent their proofs following the Curry-Howard correspondence [72] and the Brouwer–Heyting–Kolmogorov interpretation [53]. Both ideas originate from *intuitionistic logic* [53],

$$\frac{}{[\quad]well\text{-}formed}\ (Empty)$$

$$\frac{\Gamma \vdash A : Type}{\Gamma[x : A]well\text{-}formed}\ (ObjectDeclaration^{\dagger})$$

$$\frac{\Gamma \vdash A : Kind}{\Gamma[x : A]well\text{-}formed}\ (TypeDeclaration^{\dagger})$$

$$\frac{\Gamma well\text{-}formed}{\Gamma \vdash Type : Kind}\ (Type)$$

$$\frac{\Gamma well\text{-}formed \quad x : A \in \Gamma}{\Gamma \vdash x : A}\ (Variable)$$

$$\frac{\Gamma \vdash A : Type \quad \Gamma[x : A] \vdash B : Type}{\Gamma \vdash \Pi x : A, B : Type}\ (Product1)$$

$$\frac{\Gamma \vdash A : Type \quad \Gamma[x : A] \vdash B : Kind}{\Gamma \vdash \Pi x : A, B : Kind}\ (Product2)$$

$$\frac{\Gamma \vdash A : Type \quad \Gamma[x : A] \vdash B : Type \quad \Gamma[x : A] \vdash t : B}{\Gamma \vdash \lambda x : A, t : \Pi x : A, B}\ (Abstraction1)$$

$$\frac{\Gamma \vdash A : Type \quad \Gamma[x : A] \vdash B : Kind \quad \Gamma[x : A] \vdash t : B}{\Gamma \vdash \lambda x : A, t : \Pi x : A, B}\ (Abstraction2)$$

$$\frac{\Gamma \vdash t : \Pi x : A, B \quad \Gamma \vdash s : A}{\Gamma \vdash (ts) : (s \backslash x)B}\ (Application)$$

$$\frac{\Gamma \vdash A : Type \quad \Gamma \vdash B : Type \quad \Gamma \vdash t : A}{\Gamma \vdash t : B}\ (Conversion1^{\ddagger})$$

$$\frac{\Gamma \vdash A : Kind \quad \Gamma \vdash B : Kind \quad \Gamma \vdash t : A}{\Gamma \vdash t : B}\ (Conversion2^{\ddagger})$$

†: where x is not yet declared in Γ , ‡: where $A \equiv_{\beta} B$

Figure 7: Typing Rules of the $\lambda\Pi$-Calculus

which identifies truth with our ability to derive a proof of a statement in a constructive manner. This means that a concrete witness of a proof must exist, explicitly rejecting the idea that a statement can be proven by demonstrating that its negation is false. In the Curry-Howard correspondence, also known as the propositions-as-types principle, proving a statement is achieved through the construction of a term with a type corresponding to the statement, which is a direct realization of the ideas behind intuitionism. Another principle of intuitionistic logic was formulated by Brouwer and his students and proposes that the logical connectives used in a statement determine a way to compute its proof [36]. If, for instance, $A \Rightarrow B$ is to be proven, a construction needs to be provided that yields a proof of B when given a proof of A. Curry observed the applicability of this idea to the propositions-as-types principle [29] through the interpretation of connectives as type formers: Here, a proof of $A \Rightarrow B$ is identified with a function type that maps a proof of A (which is a term of type A) to one of B. Howard extended this to other connectives, particularly to the dependent type, which corresponds to quantification [44]. A more detailed overview of the development of both principles can be found in [72]. In Dedukti, rewrite rules are used to identify logical connectives with type constructors. It has been shown in [28] that the $\lambda\Pi$-calculus modulo theory is capable of expressing all *functional pure type systems* [8]. These generalize existing HOL type systems and express them within a common framework by adding different typing rules to those of simple type theory. This yields the systems of the so-called lambda cube, which illustrates the dependencies and inclusion relationships among various type systems, ranging from simple type theory to the calculus of constructions [27]. A correspondence and an embedding of the proofs of less expressive systems, as propositional or first-order logic, into pure type systems, is also given. Even though the expression of logics as pure type systems is not always used when embedding systems in the $\lambda\Pi$-calculus modulo, the fact that it is capable of expressing all functional pure type systems demonstrates its expressiveness and thus makes it an excellent candidate for a universal framework capable of encoding proofs of different theorem proving systems.

2.3.3 Decidability of Typing

As a result of the interpretation of propositions-as-types, proof checking in this encoding is reduced to type checking. This makes the decidability of type checking a central issue in this approach. Type checking is undecidable in general but can be shown to be decidable if the relation $\rightarrow_{\beta\mathcal{R}}$ is *confluent* and *terminating* [6]. These properties depend on the introduced rewrite rules and therefore cannot be proven once and for all but have to be considered individually for every system encoded in Dedukti [6].

Confluence guarantees that for any term that can be reduced in more than one way, there exists one term that the resulting terms can in turn be reduced to. As a consequence, any term can have at most one irreducible form.

Definition 1 (Confluence)**.** *let $s \rightarrow_{\beta\mathcal{R}} s'$ denote a reduction sequence from s to s'. We then call $\rightarrow_{\beta\mathcal{R}}$* ***confluent*** *if it satisfies*

$$\forall t, t_1, t_2 \, (t \rightarrow^*_{\beta\mathcal{R}} t_1 \wedge t \rightarrow^*_{\beta\mathcal{R}} t_2) \Rightarrow (\exists t' \, (t_1 \rightarrow^*_{\beta\mathcal{R}} t' \wedge t_2 \rightarrow^*_{\beta\mathcal{R}} t'))$$

In order to account for anonymous functions in the left-hand side of rewrite rules, the notion of confluence used here is defined with respect to rewriting modulo β-equivalence. This higher-order rewriting is generally undecidable, but restricting the left-hand side of the rules to Miller's patterns [52] ensures decidability in Dedukti [6].

Termination proposes that the reduction process terminates after a finite number of steps. As a consequence, any term must have at least one irreducible form. Together, these two properties hence guarantee the existence of a normal form for any term, which can be used to compare terms with regard to the congruence arising from β-reduction and rewriting, thus making type checking decidable. An important prerequisite for termination is *subject reduction* that guarantees that reductions preserve the types of terms.

Definition 2 (Subject Reduction)**.** *$\rightarrow_{\beta\mathcal{R}}$ is* ***type preserving*** *if it satisfies*

$$\forall \Gamma, t, t', T \, (\Gamma \vdash t : T \wedge t \rightarrow_{\beta\mathcal{R}} t') \Rightarrow (\Gamma \vdash t' : T)$$

Subject reduction follows if a reduction relation fulfills product compatibility and is well-typed [6]. The former arises from confluence, the latter is defined as follows.

Definition 3 (Well-Typedness [6])**.** *Let* $l \to^{\Delta} r$ *be a rewrite rule in the context* Γ_0 *and let* Γ *be a well-formed extension of* Γ_0*. We call* $l \to^{\Delta} r$ ***well-typed*** *for* Γ *if for any substitution* σ *binding the variables of* Δ*,*

$$\Gamma \vdash l\sigma : T, then \Gamma \vdash r\sigma : T.$$

3 Fundamental Encoding of Leo-III

The the blueprint the encoding of proofs in the Dedukti framework generally follows [6] will be presented here. The individual steps of the encoding process are illustrated in Figure 8.

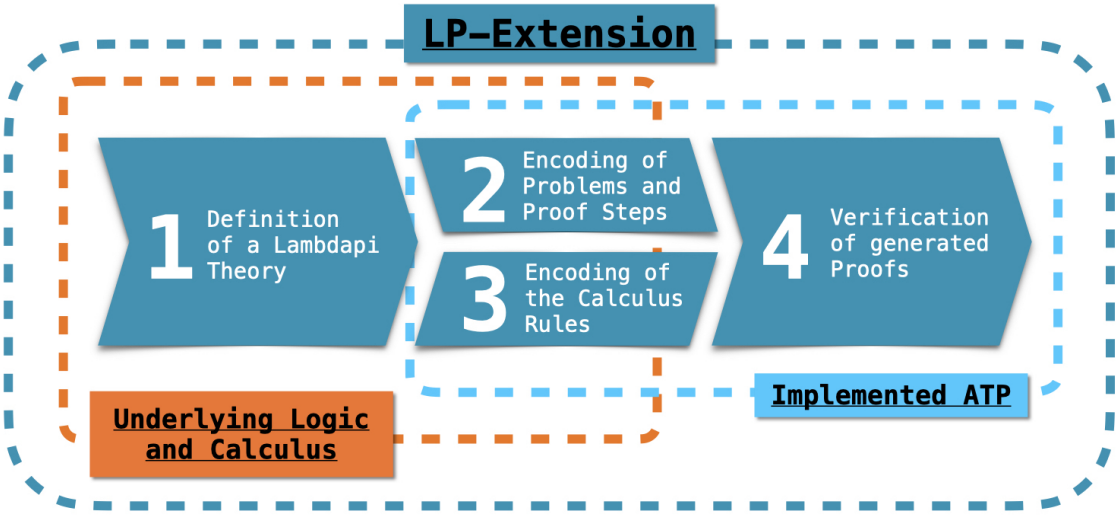

Figure 8: Steps in the encoding of ATP proofs in Lambdapi. The arrow-shaped turquoise fields divide the individual parts of the encoding into steps and their positioning in areas surrounded by dotted lines indicates the factors that need to be regarded in the encoding.

The first step in any Lambdapi encoding is the definition of a so-called Lambdapi theory. This is a set of declarations and definitions that accurately expresses the object-logic implemented by the system in terms of the meta-logic (step 1 in Fig. 8) adhering to the propositions-as-types principle. This will be discussed in Sect. 3.2 for the case of ExTT. The formulas making up a reasoning problem, as well as derived clauses representing intermediate steps of the proof, can then be encoded in terms of this logic (step 2 in Fig. 8), which is discussed in Sect. 3.3. The inference rules are encoded (step 3 in Fig. 8) by expressing the operations they perform in terms of the meta-logic of Lambdapi. Terms representing the rules must then in turn be proven. Here, these proofs are formulated using the rules of natural deduction, which correspond to the encoding of proofs as lambda-terms, as demonstrated in Sect. 3.4.1. The last step (step 4 in Fig. 8) is the construction of a complete Lambdapi-proof based on the proof that is to be verified. To this end, each step of the original proof is retraced in the Lambdapi encoding as an application of the encoded inference rules. This is arguably the most complicated part of the encoding since not only the inference rules but also their concrete implementation in Leo-III must be taken into account. The general approaches that can be employed in proof encodings are discussed in Sect. 4, and Sect. 5 is dedicated to the development of an encoding of rules and proofs in the case of Leo-III.

3.1 Lambdapi Syntax

In the following, the encodings will be given directly in the syntax of Lambdapi. A brief overview of the syntax relevant in this specific project is therefore provided here. A detailed introduction can be found in the Lambdapi user manual[6].

Terms. The syntax employed to encode terms in Lambdapi is very intuitive since it follows the familiar use of operators and notation: Abstraction, application, dependent and function types are respectively

[6] https://lambdapi.readthedocs.io/en/latest/

written `λ (x:A) y,t` (note that syntactically it is possible to provide a type, as shown for x, but it can also be obmitted, as for y), `t s`, `Π(x:A) y z, B` and `A → B`. As we will see, application in infix notation is also possible after specifying operators accordingly. Furthermore, `TYPE` is the meta-level type of types and the wildcard `_` is used for unspecified terms.

Declaring and defining symbols. A declaration of a typing in Lambdapi is done with the command `symbol` preceded by optional *modifier*s and followed by `:` and the typing.

```
modifier symbol nameOfSymbol : typeOfSymbol;
```

As previously discussed, types can be used to represent propositions in this framework. Therefore, declaring a term of a type that represents a proposition using the command `symbol` provides a proposition without a proof. Such declarations are thus referred to as axioms in Lambdapi and for clarity will be called Lambdapi-axioms in the following. There is however also a way to define a symbol with a proof-term rather than just declaring it, this is then called a Lambdapi-theorem. In Lambdapi, this is done by using `≔` after the typing and then a proof is given either in the shape of a proof-term ...

```
modifier symbol nameOfSymbol : typeOfSymbol ≔
proofTerm;
```

... or a proof-script ...

```
modifier symbol nameOfSymbol : typeOfSymbol ≔
begin
  proofScript
end;
```

Proof-terms are syntactically just regular terms while proof-scripts are constructions of proofs based on commands. The syntax and the use of the tactics is omitted here but will be given in section 3.4. The only *modifier*s used in this encoding are `constant`, that prohibits rules or definitions from being given to the symbol, `opaque`, which prohibits the terms defining theorems from ever being unfolded, and `injective`, which declares the symbol as injective.

Rules. The command for introducing rewrite rules in Lambdapi (`rule`) is followed by the left-hand side, a `↪` and the right-hand side.

```
rule nameOfHeadSymbol ... ↪ rightHandSide;
```

As discussed in section 2.3.3, rewrite rules in Dedukti and Lambdapi restrict the left-hand side to patterns to ensure the decidability of pattern matching. Rules are used to rewrite such patterns with previously declared, definable constants as their head symbols. Abstractions as well as the terms in scope and pattern variables (prefixed by `$`) can be used to form the left-hand side.

Other commands and notation. Besides `symbol` and `rule`, there are four more commands used in this project:

- `require` defines the dependency of the Lambdapi file on other files.
- `open` loads the declarations, definitions and rewrite rules of required files in the scope of the current file.

- `notation` allows to specify a symbol to have infix notation, it defines weather a symbol is left- or right-associative and defines a level of priority that determines which symbols bind more strongly in the absence of braces.
- `builtin` links declared or defined symbols to builtin ones to enable certain functionalities in proof-scripts.

Comment lines in Lambdapi are simply initiated with `\\`. It is worth mentioning that Lambdapi offers more features, for instance the cooperation of provers or the builtin use of induction, that are not discussed here because they were not used in the encoding at hand.

3.2 Encoding of ExTT as a Lambdapi Theory

A theory in the context of Lambdapi is defined as follows:

Definition 4 (Theory [18]). *Let Σ be a set of constant declarations and $\mathscr{R}$ a set of rewrite rules such that all terms occurring in the rewrite rules are declared in Σ. The pair $(\Sigma, \mathscr{R})$ is then called a **theory** if $\rightarrow_{\beta\mathscr{R}}$ is confluent and $\rightarrow_{\mathscr{R}}$ preserves typing in $(\Sigma, \mathscr{R})$.*

As discussed in section 2.3.3, these properties are essential for decidability of type checking. A universal modular theory, called *theory U*, that encompasses sub-theories corresponding to the various common logics that theorem provers are based upon is derived in [18] and available as a set of ready to use files[7]. The theory presented in the following is based upon theory U and the given definitions are from [18], even though nomenclature is adapted in some cases.

System $\mathscr{EXTT}$. As we have seen in section 2.3, there is no division of terms and types in the syntax of the meta-logic, the meta-level type `TYPE` can however be used to define the notion of object level types. Even though this work is restricted to the monomorphic version of Leo-III, a later extension to a polymorphic type system is planned. In anticipation of this, the type of object-level types is called `MonoSet`.

```
constant symbol MonoSet : TYPE;
```

It can then be used to categorize object-level monomorphic types. This way, types like o and ι can be declared:

```
constant symbol o : MonoSet;
constant symbol ι : MonoSet;
```

In order to be able to actually assign an object-level type like `o` to any declared symbol, it needs to be mapped to the meta-level `TYPE` itself. This is done with the new symbol `El`.

```
injective symbol El : MonoSet → TYPE;
```

For convenience, a symbol `Prop` is introduced for propositions. This is identified with the encoded type `El o` via a rewrite rule.

```
constant symbol Prop : TYPE;
rule El o ↪ Prop;
```

`Prop` or `El o` can then be used to declare object-level propositions like $\top$ or $\bot$.

[7] https://github.com/Deducteam/lambdapi-logics/tree/master/U

```
constant symbol ⊤ : Prop;
constant symbol ⊥ : Prop;
```

In order to encode function types like $o \to o$, a new symbol that maps two `MonoSet` to another `MonoSet` is inroduced.

```
constant symbol ⇝ : MonoSet → MonoSet → MonoSet; notation ⇝ infix right 10;
```

With this, `El (o⇝o)` would for instance encode $o \to o$. A rewrite rule is used to identify such encoded object-level function types with the meta-level function types.

```
rule El ($x ⇝ $y) ↪ El $x → El $y;
```

The logical operators of the object-logic can then be typed using `Prop` and the type constructor `⇝` analogously to the object-level type assignments.

```
constant symbol ⇒ : Prop → Prop → Prop; notation ⇒ infix right 10;
constant symbol ¬ : Prop → Prop; notation ¬ prefix 40;
constant symbol ∧ : Prop → Prop → Prop; notation ∧ infix right 30;
constant symbol ∨ : Prop → Prop → Prop; notation ∨ infix right 20;
constant symbol ⇒ : Prop → Prop → Prop; notation ⇒ infix right 10;
constant symbol ∀ [a : MonoSet] : (El a → Prop) → Prop; notation ∀ quantifier;
constant symbol ∃ [a : MonoSet] : (El a → Prop) → Prop; notation ∃ quantifier;
```

This already makes it possible to formulate object-level propositions as meta-level terms of type `Prop`. These propositions can however not yet be used to type terms, which is necessary to implement the propositions-as-types principle. To this end, the symbol `Prf` as a function mapping propositions to `TYPE` is introduced.

```
symbol Prf : Prop → TYPE;
```

As discussed previously, the Curry-Howard correspondence is encoded by identifying a proof of an implication with a meta-level function type via a rewrite rule ...

```
rule Prf ($x ⇒ $y) ↪ Prf $x → Prf $y;
```

... and identifying universal quantification with dependent types.

```
rule Prf (∀ $p) ↪ Π x, Prf ($p x);
```

These two are the primitive connectives of the system, and the remaining connectives are defined with respect to them following Russell.

```
rule Prf ⊤ ↪ Π r, Prf r → Prf r;
rule Prf ⊥ ↪ Π r, Prf r;
rule Prf (¬ $p) ↪ Prf $p → Π r, Prf r;
rule Prf ($p ∧ $q) ↪ Π r, (Prf $p → Prf $q → Prf r) → Prf r;
rule Prf ($p ∨ $q) ↪ Π r, (Prf $p → Prf r) → (Prf $q → Prf r) → Prf r;
rule Prf (∃ $p) ↪ Π r, (Π x, Prf ($p x) → Prf r) → Prf r;
```

Theory U provides no notion of equality, which is however necessary to encode the extensional type theory of Leo-III. The following symbols are therefore added as an extension to theory U. Equality is encoded with the classical interpretation due to Leibniz [50], that considers two entities possessing the same properties to be equal.

```
symbol = [T: MonoSet] : El T → El T → Prop; notation = infix right 40;
rule Prf ($x = $y) ↪ Π p , (Prf(p $x) → Prf(p $y));
```

The type `T` here is given in square brackets `[ ]`. This identifies the type to be implicit, meaning that in cases where the instantiation of `T` is clear from the context, the explicit application of `T` can be omitted. Whenever a type defined as implicit is explicitly used to instantiate a term, it is likewise enclosed in square brackets. Notions of functional and propositional extensionality are also necessary in an encoding of ExTT and are encoded by declaring them as Lambdapi-axioms.

```
symbol propExt x y : (Prf x → Prf y) → (Prf y → Prf x) → Prf (x = y);
symbol funExt [T S] (f g : (El(T ⇝ S))):Prf(∀(λ x, (f x) = (g x))) → Prf(f = g);
```

Furthermore, the logic presented thus far is constructive and does not incorporate any classical principles [53]. This includes double negation elimination, which asserts that a double negation cancels itself, and the law of the excluded middle, which states that either a statement or its negation must hold. Such classical principles can be introduced as Lambdapi-axioms [22, 5], which is here done for excluded middle. As we will demonstrate later on, double negation elimination can then be derived.

```
symbol em x : Prf(x ∨ ¬ x);
```

Definition 5 (System $\mathcal{EXTT}$)**.** *The pair $(\Sigma, \mathcal{R})$ of the symbols and rules declared herein is defined as* ***System*** $\mathcal{EXTT}$*.*

Confluence and type preservation. As discussed earlier, confluence and type preservation are properties of the relation $\rightarrow_{\beta\mathcal{R}}$ and as such need to be considered for the rewrite rules defined in each system encoded in Dedukti individually. The rewrite systems of theory U and all fragments of it are *orthogonal*.

Definition 6 (Orthogonality [7])**.** *A rewrite-system is called* ***orthogonal*** *if it is left-linear and has no critical pairs.*

Definition 7 (Critical Pairs [7])**.** *Let $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ be two rewrite rules whose variables have been renamed such that $Var(l_1, r_1) \cap Var(l_2, r_2) = \emptyset$. If there exists a $p \in Pos(l_1)$ and a most general unifier (mgu) Θ such that $l_1|_p$ is not a variable and $\Theta(l_1|_p) = \Theta(l_2)$, then a* ***critical pair*** *is given by $\langle \Theta(r_1), (\Theta(l_1)[\Theta(r_2)]_p) \rangle$.*

Intuitively, a critical pair marks an overlap in the left-hand sides of rewrite rules that can lead to situations in which one term can be rewritten in two different ways, causing ambiguity.

Definition 8 (Left Linear [7])**.** *A rewrite rule is called* ***left-linear*** *if no variable occurs more than once on the left-hand side.*

Orthogonal higher-order rewrite systems are confluent [47], thus the part of the encoding corresponding to theory U is confluent as well. Only one rewrite rule was added, namely the rewrite rule for equality

```
rule Prf ($x = $y) ↪ Π p , (Prf(p $x) → Prf(p $y));
```

It is easy to see that this rule is left-linear, since both variables of the left-hand side (`x` and `y`) occur only once, and that the left-hand side of the rule cannot form critical pairs with any other rewrite rule of the system. The derived system is thus confluent.

The second property that has been shown for theory U but remains to be verified for $\mathscr{EXTT}$ is subject reduction. As discussed previously, subject reduction follows from confluence, which has already been addressed, and well-typedness of the rewrite rules. The well-typedness of rewrite rules is not only checked automatically upon the definition of rules in Lambdapi, but can easily be ensured following [17]: Any rewrite rule where the left-hand side and the right-hand side share the same type are inherently well-typed. As demonstrated in [18], this is the case for all of the theory U rewrite rules used here and it also holds for the rewrite rule for equality.

Based on these two results and the definition of a theory, we can deduce the following theorem:

Theorem 1. *$\mathscr{EXTT}$ is a theory.*

3.3 Embedding of Problems

An embedding of a problem results in a meta-logic context containing all of the types and rules declared above ($\mathscr{EXTT}$) as well as the declarations of the object-level constant and type symbols and the formulas of the problem (i.e. the axiom and the negated conjecture). These translations are given below in terms of the Lambdapi theory representing the object-logic. Note that the encoding of types and propositions is done in two steps: The first step maps them to terms in the meta-logic, this is a *deep embedding*. The second step makes these embeddings shallow by linking the embedded term to structures of the meta-logic corresponding to their role in the object-logic [22]. A shallow embedding is desirable in a logical framework since, in contrast to the case where every object-logic defines its own constants for all structures, it establishes a common ground for different object logics [22, 6].

Definition 9 (Type Embedding)**.** *The embedding of a type as a term is denoted as $|.|$.*

$$
\begin{aligned}
|o| &= \texttt{o} \\
|\iota| &= \texttt{ι} \\
|B| &= \texttt{B}^{\dagger} \\
\underbrace{|A_1 \to ... \to A_n|}_{n} &= \underbrace{|A_1| \rightsquigarrow ... \rightsquigarrow |A_n|}_{n}
\end{aligned}
$$

The embedding $||A||$ of A as a type is given by

$$\|A\| = \texttt{El}\ |A|$$

†: Where a B is declared as defined in Def. 10

Definition 10 (Language Embedding)**.** *Let $\mathscr{L}$ be a language of HOL and T_i and t_i fresh constants for every i. The **language encoding** $\|\mathscr{L}\|$ is then a set containing a constant declaration T_i `:MonoSet` for any type constant $T_i \in \mathscr{L}$ other than o and ι and a declaration $t_i : \|T\|$ for any object-level term constant $t_i \in \mathscr{L}$.*

Definition 11 (Term Embedding)**.** *Let x be a variable and c a constant. The encoding of object-level terms, denoted as $|.|$, is then defined inductively as follows*

$$
\begin{aligned}
|c| &= \texttt{c}^{\dagger} \\
|x| &= \texttt{x} \\
|\top| &= \top
\end{aligned}
$$

$$
\begin{aligned}
|\bot| &= \bot \\
|t\, s| &= |t|\,|s| \\
|\lambda x_T, t| &= \lambda\ (x{:}\|T\|),|t| \\
|t \Rightarrow s| &= |t| \Rightarrow |s| \\
|\forall x_T.t| &= \forall(\lambda\ (x{:}\|T\|),|t|) \\
|\exists x_T.t| &= \exists(\lambda\ (x{:}\|T\|),|t|) \\
|\neg t| &= \neg|t| \\
|t \vee s| &= |t| \vee |s| \\
|t \wedge s| &= |t| \wedge |s| \\
|t = s| &= |t| = |s|
\end{aligned}
$$

This results in the following encoding for clauses and literals of equational and non-equational form:

$$
\begin{aligned}
|l_1 \vee ... \vee l_n| &= \mathtt{Prf}\ |l_1 \vee ... \vee l_n| \\
|[s \simeq t]^{tt}| &= \mathtt{Prf}\ s = t \\
|[s \simeq t]^{ff}| &= \mathtt{Prf}\ |\neg(s = t)| \\
|[s]^{tt}| &= \mathtt{Prf}\ |s| \\
|[s]^{ff}| &= \mathtt{Prf}\ |\neg s|
\end{aligned}
$$

†: *Where* `c` *is declared as defined in Def. 10*

Based on the term encodings, the terms of type o can be embedded as propositions.

Definition 12 (Problem Embedding)**.** *A proposition γ is encoded as*

$$\|\gamma\| = \mathtt{Prf}\ |\gamma|$$

Let $\Delta = \{\gamma_1, ...\gamma_n\}$ be the set of axioms of a HOL reasoning problem, $x_{i,1}, ..., x_{i,m}$ the free variables of γ_i and ax_i a fresh constant for every i. Δ is then embedded as follows

$$\|\Delta\| = ax_1 {:}\Pi x_{1,1} ... \Pi x_{1,m}\|\gamma_1\|,\ \ldots,\ ax_n {:}\Pi x_{n,1} ... \Pi x_{n,m}\|\gamma_n\|$$

3.4 Encoding of Proofs

As previously discussed, the Curry-Howard correspondence and the Brouwer–Heyting–Kolmogorov interpretation treat propositions-as-types and logical connectives as type constructors. Specifically, implication is associated with the function type constructor. Howard first pointed out the natural extension of this correspondence with the coherence between the rules of *natural deduction* (ND) [56], a proof calculus mirroring human intuition developed by Gentzen [38], and term operations. The calculus consists of introduction and elimination rules that capture the interpretations of the logical connectives by defining the conditions under which they can be added or removed from formulas. Consider, for instance, the rules for implication:

$$\dfrac{\begin{array}{c}[x]\\ y\end{array}}{x \Rightarrow y}\Rightarrow\text{-I} \qquad \dfrac{x \Rightarrow y \qquad x}{y}\Rightarrow\text{-E}$$

Here, the square brackets [.] signify that a term is an assumption. In natural deduction, assumptions can be made freely and used in deduction, but they must be *discarded* in the proof. This is achieved through rules such as ⇒-I, which states that the implication $x \Rightarrow y$ can be inferred if y can be proven based on the assumption x. This rule captures the interpretation of implication by allowing hypothetical reasoning: When assuming that x leads to y, then $x \Rightarrow y$ holds true. This method preserves the character of x as an assumption and ensures that the conclusion y is conditionally derived from x. ⇒-E (Modus Ponens) then allows concluding y if the premise x is given. To see the correspondence of these rules with term operations, we recall the interpretation of λ-terms as proofs and of their types as the proven propositions. Here, the Lambdapi notation previously introduced is used, but of course, this correspondence holds in all systems implementing the propositions-as-types principle. Consider for instance `t : Prf a`, a term representing the proof of some proposition `a`. Since we have a proof of `a`, we should also be able to construct a proof of `b ⇒ a` for any proposition `b`, which would result in type `Prf(b ⇒ a)`. In the ND proof, ⇒-I would allow this conjecture, but how is it mirrored in the encoding as types? As we have seen, `Prf(b ⇒ a)` is identified with the function type `Prf b → Prf a`, therefore proving the implication is equivalent to constructing a term based on `t` that has this function type. This can be achieved with λ-abstraction over terms that themselves represent proofs: `λ x : Prf b, t` is a proof-term of the wanted type. Therefore, ⇒-I corresponds to type abstraction. Similarly, application realizes ⇒-E since it applies an argument of type `Prf b` to a term of type `Prf b → Prf a` and results in a term typed with `Prf a`.

The second logical connective we interpreted in terms of a type constructor was universal quantification, the introduction and application rules for which are...

$$\dfrac{A(y)}{\forall x\, A(x)}\forall\text{-I}^{\dagger} \qquad \dfrac{\forall x\, A(x)}{A(y)}\forall\text{-E}^{\ddagger}$$

†: Where y is not free in any undischarged hypothesis and x is fresh
‡: Where y is not a bound variable in any other term in the derivation

Like in the case of implication, the rules for quantification can be interpreted in terms of abstraction and application, but this time we abstract over terms encoding objects of the logic rather than proofs: Abstracting over a variable that is free in the type of a term will result in a dependent type, which is associated with quantification. We can for instance extend the proof-term we constructed before to express that the implication `Prf(b ⇒ a)` holds for arbitrary b through an additional abstraction: The term `λ (b : Prop) (x : Prf b), t` has type `Π b: Prop, Prf b → Prf a`. Instantiating such terms through application, in this case of a proposition, corresponds to ∀-E.

Since we treated implication and universal quantification as the primitive connectives and defined all other connectives with respect to them using rewrite rules, the construction of proof-terms can be understood in this light. If we try to find an encoding of a proof for propositions like $\forall a.(a \Rightarrow (a \vee a))$ this way, we thus can rewrite logical connectives like $\vee$. Here this results in $\forall a.(a \Rightarrow (\forall b.(a \Rightarrow b) \Rightarrow (a \Rightarrow b) \Rightarrow b))$. A proof of this in natural deduction is given in the tree notation of Gentzen:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[x \Rightarrow y] \qquad \dfrac{[x \Rightarrow y] \qquad [x]}{y}\Rightarrow\text{-E}}{(x \Rightarrow y) \Rightarrow y}\Rightarrow\text{-I}}{(x \Rightarrow y) \Rightarrow (x \Rightarrow y) \Rightarrow y}\Rightarrow\text{-I}}{\forall b_o.(x \Rightarrow b) \Rightarrow (x \Rightarrow b) \Rightarrow b}\forall\text{-I}}{x \Rightarrow (\forall b.(x \Rightarrow b) \Rightarrow (x \Rightarrow b) \Rightarrow b)}\Rightarrow\text{-I}}{\forall a.a \Rightarrow (\forall b.(a \Rightarrow b) \Rightarrow (a \Rightarrow b) \Rightarrow b)}\forall\text{-I}$$

We can now construct a λ-term that encodes this proof, i.e. a term of type `Π a: Prop, Prf a →` `Π b: Prop, (Prf a → Prf b) → (Prf a → Prf b) → Prf b`. For each of the assumptions in the proof we will need to abstract over a term encoding a proof of them, we will thus assume three terms of the types `h1: Prf a`, `h2: Prf a → Prf b` and `h3: Prf a → Prf b`. The body of the λ-term then must be of the return type, in this example `Prf b`. We get such a term by applying `h1` to `h2`, this corresponds to the first rule of the ND proof: the ⇒-E step. We then need to abstract over the assumptions, this corresponds to the introduction rules in the ND proof and will result in function- and dependent types, corresponding to implication and universal quantification introduction. The first abstraction encodes the second step of the proof, which applies ⇒-I and discards $a \Rightarrow b$. The following steps can then also be encoded through abstraction in the same way. This results in a proof-term that we can use to define the theorem in a Lambdapi encoding:

```
symbol ex_1_1 : Π a: Prop, Prf a → Prf (a ∨ a)
≔ λ a (h1:Prf a) b (h2:(Prf a → Prf b)) (h3:(Prf a → Prf b)), h2 h1;
```

Computation here thus corresponds to proof simplification. It is worth noting a few ways in which the notation can be simplified at this point: Types for the hypothesis can be omitted if they are unambiguous and variables in abstractions representing ⇒-I can be replaced with `_` if the hypothesis is never used by elimination rules. A simpler version of the term (that would still be accepted by Lambdapi) would therefore be...

```
opaque symbol ex_1_1_simp : Π a: Prop, Prf a → Prf (a ∨ a)
≔ λ a h1 b h2 _, h2 h1;
```

3.4.1 Deriving the Rules of Natural Deduction

Natural deduction does not only offer rules for implication and unification. The introduction rules of ∨ for instance allow to conclude $x \vee y$ if either x or y are proven.

$$\frac{x}{x \vee y}\ \vee\text{-I}_r \qquad\qquad \frac{y}{x \vee y}\ \vee\text{-I}_l$$

Proving $\forall a.a \Rightarrow (a \vee a)$ becomes more straight-forward when using $\vee$-I$_r$:

$$\dfrac{\dfrac{\dfrac{[b]}{b \vee b}\ \vee\text{-I}_r}{b \Rightarrow (b \vee b)}\ \Rightarrow\text{-I}}{\forall a.a \Rightarrow (a \vee a)}\ \forall\text{-I}$$

We can encode such rules as functions that take proofs of the premises and return a proof of the conclusion. This then itself becomes a proposition we can construct a proof-term for by assuming the premise, in the case of ∨-Ir `Prf x`, and proving the conclusion, here `Prf(x ∨ y)` (which rewrites to `Π r: Prop, (Prf x → Prf r) → (Prf y → Prf r) → Prf r`):

```
opaque  symbol ∨Ir : Π x: Prop, Π y: Prop, Prf y → Prf (x ∨ y)
≔ λ x y h1 b _ h3, h3 h1;
```

The proof of $\vee$-I$_l$ is analogous. These rules can then be instantiated and used in Lambdapi proofs as constants. The following proof for instance instantiates the rule with `a a`, resulting in a term of type `Prf(a) → Prf(a ∨ a)`, which represents a proof of the previous example.

```
symbol ex_1_2_term:  Π a, Prf(a) → Prf(a ∨ a):=
λ a, ∨Ir a a
```

The other rules of natural deduction, as well as the truth of $\top$, the implication of arbitrary propositions by $\bot$ and functions representing the reflexively and the definition of equality can also be encoded this way and proven in terms of the rewriting defined for the connectives:

Natural Deduction rules

$$\frac{\begin{array}{c}[x]\\ y\end{array}}{x \Rightarrow y}\ \Rightarrow\text{-I}$$

```
opaque symbol ⇒I : Π x: Prop, Π y: Prop, (Prf x →
    Prf y) → Prf (x ⇒ y)
:= λ x y h1 h2, h1 h2;
```

$$\frac{x \Rightarrow y \qquad x}{y}\ \Rightarrow\text{-E}$$

```
opaque symbol ⇒E : Π x: Prop, Π y: Prop, Prf (x ⇒ y)
    → Prf x → Prf y
:= λ x y h1 h2, h1 h2;
```

$$\frac{A(y)}{\forall x\ , A(x)}\ \forall\text{-I}^{\dagger}$$

```
opaque symbol ∀I : Π T: MonoSet, Π p: (El T → Prop),
    (Π x: El T, Prf (p x)) → Prf (‘∀ y, p y)
:= λ T p h1, h1;
```

$$\frac{\forall x\, A(x)}{A(y)}\ \forall\text{-E}^{\ddagger}$$

```
opaque symbol ∀E : Π T: MonoSet, Π p: (El T → Prop),
    Prf (‘∀ y, p y) → Π x: El T, Prf (p x)
:= λ T p h1, h1;
```

$$\frac{x \vee y \qquad \begin{array}{c}[x]\\ z\end{array} \qquad \begin{array}{c}[y]\\ z\end{array}}{z}\ \vee\text{-E}$$

```
opaque symbol ∨E : Π x: Prop, Π y: Prop, Π z: Prop,
    (Prf x → Prf z) → (Prf y → Prf z) → Prf (x ∨ y)
    → Prf z
:= λ x y z h1 h2 h3, h3 z h1 h2;
```

$$\frac{x \qquad y}{x \wedge y}\ \wedge\text{-I}$$

```
opaque symbol ∧I : Π x: Prop, Π y: Prop, Prf x → Prf
    y → Prf (x ∧ y)
:= λ x y h1 h2 b h3, h3 h1 h2;
```

$$\frac{x \wedge y}{x}\ \wedge\text{-E}_l$$

```
opaque symbol ∧El : Π x: Prop, Π y: Prop, Prf (x ∧ y
    ) → Prf x
:= λ x y h1, h1 x (λ h2 _, h2);
```

$$\frac{x \wedge y}{y}\ \wedge\text{-E}_r$$

```
opaque symbol ∧Er : Π x: Prop, Π y: Prop, Prf (x ∧ y
    ) → Prf y
:= λ x y h1, h1 y (λ _ h3, h3);
```

$$\frac{\begin{array}{c}[x]\\ \bot\end{array}}{\neg x}\ \neg\text{-I}$$

```
opaque symbol ¬I : Π x: Prop, (Prf x → Prf ⊥) → Prf
    (¬ x)
:= λ x h1 h2, h1 h2;
```

$\frac{\neg x \qquad x}{\bot}$ ¬-E

```
opaque symbol ¬E : Π x: Prop, Prf x → Prf (¬ x) →
    Prf ⊥
:= λ x h1 h2, h2 h1;
```

†: Where y is not free in any undischarged hypothesis and x is fresh
‡: Where y is not a bound variable in any other term in the derivation

Additional rules

$\frac{}{\top}$ ⊤-I

```
opaque symbol ⊤I : Prf ⊤
:= λ b h1, h1;
```

$\frac{\bot}{x}$ ⊥-E

```
opaque symbol ⊥E: Π a: Prop, Prf ⊥ → Prf a
:= λ a h1, h1 a;
```

$\frac{x = y}{\forall p.(p\,y) \Rightarrow (p\,x)}$ =-def

```
opaque symbol =def : Π [T: MonoSet], Π x: El T, Π y:
     El T, Prf (x = y) → Π p: (El T → El o), Prf (p
    y) → Prf (p x)
:= λ T x y h1 p h2, h1 (λ z, z = x) (λ p2 h3, h3) p
    h2;
```

$\frac{}{x = x}$ =-ref

```
opaque symbol =ref : Π [T: MonoSet], Π x: El T, Prf
    (x = x)
:= λ T x p h, h;
```

Furthermore, the principle of double literal elimination can be derived using the principle of excluded middle and the rules of natural deduction:

Double literal elimination

$\frac{\neg\neg x}{x}$ npp

```
opaque symbol npp x : Prf(¬ ¬ x) → Prf x :=
begin
    assume x h1;
    refine ∨E x (¬ x) x _ _(em x)
        {assume h2;
        refine h2}
        {assume h2;
        refine ⊥E x (¬E (¬ x) h2 h1)}
end;
```

3.5 Correctness of the Encoding

There are two important characteristics that translations must fulfill in order to result in a useful encoding of proofs: They must be complete, meaning that any embedded type, term and proof must be typed

correctly, and they must be sound, ensuring that any proof derived in an encoding actually corresponds to a correct proof in the original system. Together these two properties guarantee *correctness*.

Definition 13 (Correctness)**.** *Let $\mathcal{O}\mathscr{L}$ be the theory encoding an object-logic and $\mathscr{L}$ be a language of the object-logic. An embedding is called **correct** if, for any set of assumptions Δ and any formula δ expressed in $\mathscr{L}$, there exists a λ-term t such that*

$$\mathcal{O}\mathscr{L}, \|\mathscr{L}\|, \|\Delta\| \vdash t : \|\delta\| \quad \textit{if and only if} \quad \Delta \vdash \delta \textit{ is provable in the object-logic.}$$

Theorem 2 (Completeness)**.** *Let $\mathscr{L}$ be a language of HOL. For any proof $\mathscr{P}$ of $\Delta \vdash \delta$ in ND there exists a λ-term t that satisfies*

$$\mathcal{EXTT}, \|\Delta\| \vdash t : \|\delta\| \quad \textit{if and only if} \quad \Delta \vdash \delta \textit{ is provable in ExTT.}$$

Proof. By structural induction over the proof $\mathscr{P}$ and the encoded propositions, terms and types. □

While soundness is proposed, no proof for theory U or the used fragment exists to date and is left for future work. Related work and possible approaches are however discussed in Sect. 7.3.

3.5.1 Proof-Scripts

As an alternative to proof-terms, Lambdapi proofs can be given in the form of interactive scripts using the keywords `begin` and `end`. These take the type to be constructed as the so-called *focused goal* and allow constructing proofs step by step. If we for instance begin a proof-script for the previous example (`ex_1_2`), the focused goal would be `Π a: Prop, Prf a → Prf (a ∨ a)` . Since each step can be checked individually and because scripts result in a more illustrative representation, they are very useful in the development of proofs. Furthermore, proof-scripts offer some tactics that can facilitate deriving proofs: `assume` and `refine` can be used to directly express any proof formulated as a λ-term. `assume` corresponds to abstraction, i.e. the introduction of function types or dependent types which can therefore be removed from the typing goal while they introduce a hypothesis (in the case of function types) or instantiate the typing goal with the given variable name (in the case of dependent types). In the first step of a proof-script for `Π a: Prop, Prf a → Prf (a ∨ a)` , we could thus `assume a` and would be left with the focussed goal `Prf a → Prf (a ∨ a)` . `refine` instantiates the focused goal and thus corresponds to the elimination rules. With this we can complete the proof-script:

```
1 symbol ex_1_2_script a: Prf(a) → Prf(a ∨ a)≔
2 begin
3     assume a;
4     refine ∨Ir a a;
5 end;
```

Note that an alternative syntax for dependent types at the outermost scope of a typing omits `Π` from the typing and instead gives the variable names behind the name of the defined symbol as done in the definition above.

Representation of Subproofs. In some cases, proving a conjecture involves providing subproofs: We can, for instance, prove `Prf(a = (a ∨ a))` using `propExt` , which can be instantiated with two propositions `x` and `y` and then takes both `Prf x ⇒ Prf y` and `Prf y ⇒ Prf x` as arguments and maps them to `Prf(x = y)` . In the interactive proof-scripts, such subproofs can be solved as individual focused goals that are initiated by refining the initial goal with a `_` in place of the terms and providing the proofs in curly brackets. Similarly, the tactic `have` allows defining new variables as subproofs and typing them with the statement to be proven. Using this tactic allows us to prove arbitrary implications and thus provides us with the option to introduce assumptions that are not directly encoded in the

focused goal. This can, for instance, be handy in cases where the same proof-term is used more than once. We can, for example, use `∨E` in the subproof of `Prf(a ∨ a) → Prf a` to eliminate ∨ by showing that, naturally, both sides of the disjunction imply `a`. Both the use of the `refine` and the `have` tactics are demonstrated in the following for providing subproofs:

```
symbol ex_1_3 a: Prf(a = (a ∨ a)):=
begin
    assume a;
    refine propExt a (a ∨ a) _ _
        {refine ex_1_2_script a}
        {assume h1;
        have H1: Prf a → Prf a
            {assume h2;
            refine h2};
        refine ∨E a a a H1 H1 h1}
end;
```

Hypothesis introduced using the `have` tactic are labeled with capital letters (for instance `H1`), while those assumed (with `assume`) are labeled with lower case letters (for instance `h1`). Here it would of course be possible to provide the very same proof without subproofs, in this case the proof-terms would just be provided in the appropriate positions. The corresponding proof-term is given below.

```
symbol ex_1_3_term : Π a: Prop, Prf (a = (a ∨ a))
≔ λ a, propExt a (a ∨ a) (ex_1_3 a) (λ h1, ∨E a a a (λ h2, h2) (λ h2, h2) h1);
```

Equality Tactics. Lambdapi also offers a number of other tactics, the ones that will be of particular use for this project utilize equality. They can be used after linking the terms we defined for `Prf`, `El`, `=`, `=def`, and `=ref` to their built-in versions in Lambdapi. `reflexivity` then proves goals of the form `Prf(x = y)` if `x` and `y` are equivalent, and `symmetry` flips `x` and `y` of goals of the shape `Prf(x = y)`. Another useful tactic is `rewrite`. As we have seen, actual rewrite rules can be very useful, but one must be careful when introducing them since the confluence and termination of the system depend on them and generally they should thus only be defined in the Theory. `rewrite` provides a way to use terms encoding the proofs of equalities like rewrite rules within proof-scripts. Instead of defining actual rewrite rules, Lambdapi applies the encoded equality to construct proof-terms using `=def` and `=ref`. The reader should therefore keep in mind that the actual operations performed by rewrite rules and the `rewrite` tactic are quite different, even though, for the sake of readability, both operations will be referred to as 'rewriting' in the following. This way, the tactic allows users to use equalities as if they were rewrite rules within the corresponding step which, as we sill see, present a very convenient and user-friendly way of deriving proofs. We can, for instance, use the term of type `Π a: Prop, Prf(a = (a ∨ a))` to provide a proof for `b` based on an axiom `Prf (b ∨ b)` by rewriting the goal to `Prf (b ∨ b)` and refining:

```
symbol ex_2_axiom : Prf(b ∨ b);

symbol ex_2_conj : Prf b ≔
begin
    rewrite ex_1_3;
    refine ex_2_axiom
end;
```

Note that currently `rewrite` is not capable of rewriting subterms under binders. It is however planned to lift this restriction in future versions of Lambdapi.

3.6 Lambdapi File Structure

The `rewrite` tactic will prove to be very useful in the following. However, the rewriting of connectives to their interpretations using the rules defined earlier can prohibit the pattern matching of the rewrite tactic and thus render it useless. This problem can be met by tailoring the structure of the Lambdapi files of a proof encoding to separate the rewrite rules for the non-primitive connectives from the encoded proofs, as shown in Fig. 9.

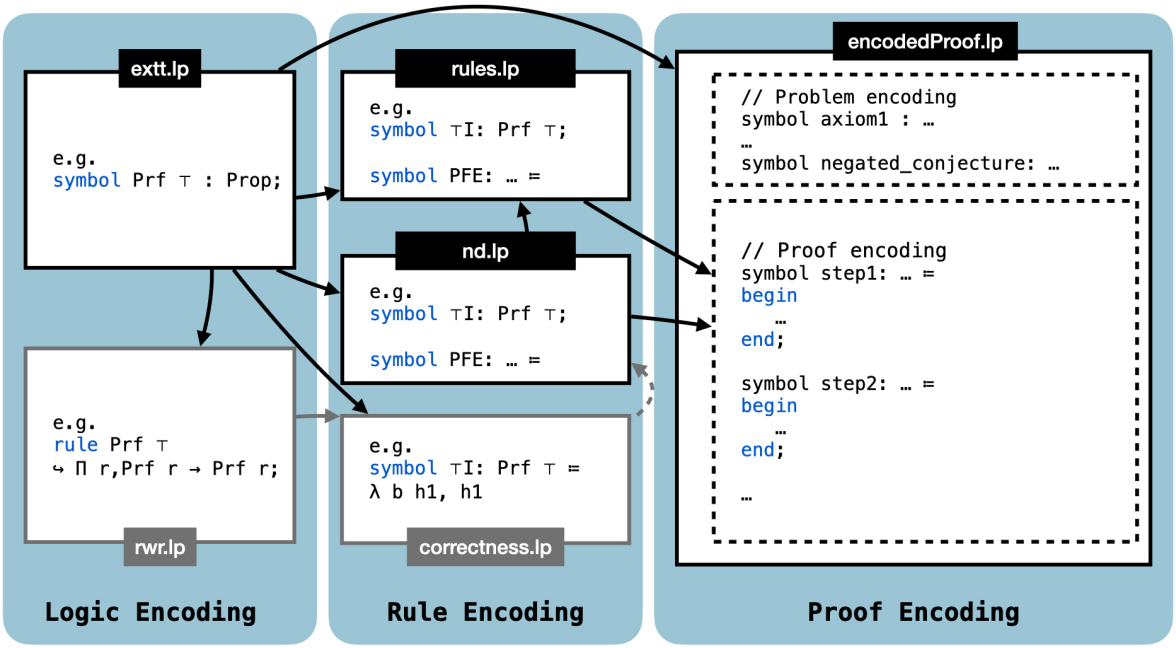

Figure 9: Structure and dependencies of files. The files containing the definition of the rewrite rules are marked in gray, the others are marked in black. Arrows with solid lines indicate that the symbols of one file are imported in another file and the dotted line indicates that the symbols defined in one file are declared in another file.

extt.lp is the Lambdapi file containing all the rules, declarations, and definitions discussed in Section 3 except for the rules of the connectives. It forms the basis of all the encodings in the other files. rwr.lp contains these rules, and correctness.lp provides the proofs of the ND-rules given in Section 3.4.1 based on the rewrite rules. The same ND encodings can then be used to encode and prove the calculus rules and the proofs of Leo-III without needing the actual rewrite rules. Therefore, they are declared (but not defined) in nd.lp and in rules.lp, the inference and accessory rules are proven based on them. encodedProof.lp then contains the encoding of the proof to be verified.

4 Common Encoding Challenges and Approaches

We have observed that, in the case of ND, inference rules can be encoded as functions operating on proof-terms. Proofs are then constructed by instantiating the functions and applying axioms to them. The encoding of the EP calculus and the proofs derived by Leo-III is more involved, as the calculus rules are more complex, and additional factors, particularly the implementation of these rules in Leo-III, must be considered. As we will discuss below, the challenges in encoding various calculus rules and proof steps are diverse and necessitate different strategies. In order to establish general methods for the encoding of these different rules, referred to as *encoding approaches* here, the following categorization was developed to classify the demands involved in verifying the application of Leo-III calculus rules:

- **Adaptability of rule encodings**
  - **Static**: The inference rule can be encoded as a single term in Lambdapi.
  - **Versatile**: The inference rule is represented by multiple encoded terms.
  - **Flexible**: A flexible on-the-fly generation of the inference rule and its proof is necessary.
- **Structures the rule operates on**
  - **Clauses**: The rule operates on whole clauses.
  - **Literals**: The rule operates on individual literals.
  - **Terms**: The rule operates on substructures of literals.
- **Implicit transformations**
  - **Transforming to equality literal**
  - **Changing the order within equality literals**
  - **Changing the order of literals**
  - **Deletion of double literals**
  - **Application of other calculus rules**

In the following, the categories will be explained in more detail, and encoding approaches for each case will be developed using both proof-terms and proof-scripts. It is worth noting that the proofs of additional rules in the 'proof-term' sections will still mostly be presented as proof-scripts, as this representation makes it much easier to comprehend the proofs. However, each of them could easily be expressed as a proof-term as well and the use of the rules in proofs using proof-terms is demonstrated for some examples.

4.1 Adaptability of rule encodings

Encoding the operations performed by inference rules can require a high level of flexibility to cover the full range of structural variations found in the clauses serving as premises. As we will see, this need for flexibility can make it impossible to encode an inference rule using only a single Lambdapi term, or it may even prohibit the use of predefined encodings altogether. This is what we refer to as the *adaptability* of a rule.

4.1.1 Static

Recall that we have encountered two methods of representing inference rules. On the one hand, we encoded them as functions mapping proofs of premises to proofs of conclusions. On the other hand, we expressed inferences as equalities, which could then be used with the `rewrite` tactic. In all cases considered so far, the rules could, however, be encoded statically, in the sense that one fixed definition sufficiently covered the different possible arguments and conclusions. An example of this was the enocoding of...

$$\frac{y}{x \vee y} \vee\text{-I}_r$$

as a term of type `Π x : Prop, Π y : Prop, Prf y → Prf (x ∨ y)`.

Since this method of encoding is generally the most straightforward, it will be employed whenever possible.

4.1.2 Versatile

In other cases, the notation used in the calculus rules allows for an ambiguity that the Lambdapi syntax cannot cover. Examples of this are rules that apply to both negative and positive literals, such as the symmetry of equality:

$$\frac{(x = y)^\alpha}{(y = x)^\alpha} =\text{-}Sym$$

To encode this as a function, we will need two versions: one for positive equality terms (`Prf (x = y) → Prf (y = x)`) and one for negative ones (`Prf (¬(x = y)) → Prf (¬(y = x))`). This represents a rule needed for the verification of Leo-III proofs, and the encoding and proof are provided in section 4.3.2.

4.1.3 Flexible

Some inferences require more flexibility than predefined encodings can provide. These rules must therefore be defined dynamically and tailored to the specific application. For instance, consider the commutativity of disjunctions, which is not inherently given in Lambdapi and must therefore be proven explicitly. This is accomplished through the permutation σ:

$$\frac{l_1 \vee ... \vee l_i \vee ... \vee l_n}{l_{\sigma(1)} \vee ... \vee l_{\sigma(i)} \vee ... \vee l_{\sigma(n)}} \vee\text{-Permut}$$

This rule must handle arbitrary permutations on clauses of varying lengths. Consequently, a function that maps the proof of a given clause to a proof of the permutation must be formulated and proven for each individual case. As this too is a rule we will need in the verification of Leo-III proofs, the proof construction is discussed in section 4.3.3.

In such cases, a proof that is flexible enough to be adapted to specific scenarios can be derived automatically. These rules can then either be defined as their own Lambdapi terms and instantiated in proofs (like any other inference rule), or their proofs can be directly integrated into the Leo-III verification steps where they are needed. The former approach is used here for rules that are not a part of the calculus EP but still necessary in the encoding, while the latter is employed for calculus rules that need to be proven on-the-fly.

4.2 Operations on Substructures

4.2.1 Operation on the whole Clause

Rules operating on clauses rather than on their substructures can simply be encoded as functions mapping proof-terms of encoded clauses as a whole to their conjectures, making them easy to apply. Recall for instance that we provided a proof for `Π a, Prf (a = (a ∨ a))` in section 3.4 by proving both `Prf a → Prf (a ∨ a))` and `Prf (a ∨ a)) → Prf a`. Analogous to the latter, we could encode the inference

$$\frac{x \vee x}{x} \text{Simp1}$$

as a Lambdapi term `simp1 x: Prf (x ∨ x) → Prf x`. Such *simplification rules* can be used to derive less complex terms from parent formulae:

```
1 symbol ex_3_axiom : Prf(b ∨ b);
2
3 symbol ex_3_conj : Prf(b)≔
4 simp1 b ex_3_axiom;
```

4.2.2 Operations on Single Literals

The application of such rules becomes more complicated if we want to use an inference like this on clauses consisting of more than just one literal. Let us for instance prove `a ∨ b` for arbitrary propositions `a` and `b` based on an axiom `ex_4_axiom : Prf(a ∨ (b ∨ b))`.

Proof-terms. Using proof-terms, this can be achieved through an additional rule that allows us to construct functions that transform only a single literal of a clause. The proof of such a function is based on a proof of the original clause and a proof of a function mapping one of the disjuncts to a new proposition. As an inference, a rule like this would take the form ...

$$\frac{x \vee y \qquad y \Rightarrow z}{x \vee z} \vee\text{-}c_r$$

We can thus construct functions that map the encodings of rules operating on single literals to functions operating on whole encoded clauses. Rules like this will therefore be called *construction rules* here. $\vee$-c_r can be encoded and proven in Lambdapi using `∨E` along with subproofs demonstrating that both disjuncts of the original term imply the new disjunction (line 4 in the proof shown below). For the left disjunct, this can simply be done using `∨Il`, since it is itself in the derived clause (line 6). The right-hand side can be proven analogously, but this time the right disjunct first has to be mapped to the new term using the inference operating on the literal (which was assumed in line 3 as `h1 : Prf y → Prf z`).

```
1 opaque symbol c∨r x y z : (Prf y → Prf z) → Prf(x ∨ y) → Prf(x ∨ z) ≔
2 begin
3     assume x y z h1 h2;
4     refine ∨E x y (x ∨ z) _ _ h2
5         {assume h3;
6         refine ∨Il x z h3}
7         {assume h3;
8         refine ∨Ir x z (h1 h3)}
9 end;
```

`c∨r` can then be instantiated and used to prove the application of `simp1` on a single literal:

```
1 symbol ex_4_conj : Prf(a ∨ b)≔
2 c∨r a (b ∨ b) b (simp1 b) ex_4_axiom;
```

This rule is however limited to clauses of length two. Constructing a proof-term of type `Prf(a ∨ b ∨ c)` based on

```
1 symbol ex_5_axiom : Prf(a ∨ (b ∨ b) ∨ (c ∨ c));
```

would therefore require a nested application of `c∨r`. It would be more convenient to be able to directly map proofs of clauses of an arbitrary length to ones where inference rules were applied to some of the literals. Rules performing this operation can be generated and proven on-the-fly. These proofs follow the same pattern shown for `c∨r` with a nested, repeated use of ∨-E. We call the inferences generated this way `transformation` rules. An automatically generated proof for a Lambdapi function taking a clause of length three and rules operating on the second and third literal is for instance given by:

```
opaque symbol transform_0_1_1 : (Π (x0 : (El o)), Π (x1 : (El o)), Π (x2 : (El o
    )), Π (x3 : (El o)), Π (x4 : (El o)), (((Prf x1) → (Prf x2)) → ((Prf x3) → (
    Prf x4)) → (Prf (x0 ∨ x1 ∨ x3)) → (Prf (x0 ∨ x2 ∨ x4)))≔
begin
    assume x0 x1 x2 x3 x4 h0 h1 h2;
  refine (∨E x0 (x1 ∨ x3) (x0 ∨ x2 ∨ x4) _ _ h2)
    {assume h3;
    refine (∨Il x0 (x2 ∨ x4) h3)}
    {assume h3;
    refine (∨E x1 (x3) (x0 ∨ x2 ∨ x4) _ _ h3)
      {assume h4;
      refine (∨Ir x0 (x2 ∨ x4) (∨Il x2 (x4) (h0 h4)))}
      {assume h4;
      refine (∨Ir x0 (x2 ∨ x4) (∨Ir x2 (x4) ((h1 h4))))}}
end;
```

Here these rules are called `transform` with a postfix encoding the positions of literals in the clause that are to be transformed (`1`) and the ones to which no transformation is to be applied (`0`).

With this rule, we can then prove the conjecture in one step:

```
symbol ex_5_1_conj : Prf(a ∨ b ∨ c)≔
transform_0_1_1 a (b ∨ b) b (c ∨ c) c (Simp1 b) (Simp1 c) ex_5_axiom;
```

Proof-scripts. An inference encoded as an equality literal can simply be applied using the `rewrite` tactic, regardless of the number of literals. The use of the rewrite tactic for simplification rules in this fashion is already used successfully in other projects [26] in the Dedukti framework, which have motivated the handling of such rules as shown here.

In `ex_1_3`, we have already proven the equality corresponding to *simp*1. It is given by...

```
symbol simp1_eq x: Prf(x = (x ∨ x));
```

We can now use `simp1_eq` to prove the conjecture of `ex_5` without having to generate any additional rules. The equality is used to rewrite the corresponding terms in the specified position:

```
symbol ex_5_2_conj : Prf(a ∨ b ∨ c)≔
begin
    rewrite .[x in _ ∨ x ∨ _] simp1_eq;
    rewrite .[x in _ ∨ _ ∨ x] simp1_eq;
    refine ex_5_axiom
end;
```

Such an encoding of a rule is of course only possible if the premise and the conclusion of a rule are actually equivalent. In cases where the conclusion is a mere logical consequence of the premise, the `transform` rule described above still needs to be applied when encoding proofs using scripts.

4.2.3 Operations on Substructures of Literals

Encoding rules that operate on substructures of literals follows the same ideas as the encoding of rules operating on single literals both in the case of proof-terms and of proof-scripts.

Proof-terms. Once again, we can access the substructures through the use of construction rules. To that end, rules like `c∨r` have to be derived for the other connectives as well. If we for instance want to construct a proof for `c ⇒ b` based on the axiom

```
symbol ex_6_axiom : Prf(c ⇒ (b ∨ b))
```

we have to encode the rule

$$\frac{x \Rightarrow y \qquad y \Rightarrow z}{x \Rightarrow z}\ \text{c}\Rightarrow$$

which is done using the introduction and elimination rules of implication:

```
opaque symbol c⇒ x y z : (Prf y → Prf z) → Prf(x ⇒ y) → Prf(x ⇒ z) ≔
begin
    assume x y z h1 h2;
    refine ⇒I x z _;
    assume h3;
    refine h1 (⇒E x y h2 h3)
end;
```

`c⇒` can then be instantiated and used to apply the simplification rule:

```
symbol ex_6_conj : Prf(c ⇒ b)≔
c⇒ c (b ∨ b) b (simp1 b) ex_6_axiom;
```

Similar construction rules are defined for the different connectives in appendix A. For more complex terms, these construction rules can be nested and used to access arbitrary substructures:

```
symbol ex_7_axiom : Prf(a ∨ (c ⇒ (b ∨ b)));

symbol ex_7_1_conj : Prf(a ∨ (c ⇒ b))≔
c∨r a (c ⇒ (b ∨ b)) (c ⇒ b) (c⇒ c (b ∨ b) b (simp1 b)) ex_7_axiom;
```

Proof-scripts. Once again, there are no additional rules necessary when using proof-scripts and equality terms, the position of the substructure simply has to be indicated by the pattern used in the `rewrite` tactic, making the nested use of construction rules superfluous. This way, the conjecture of the previous example can be proven using `simp1_eq`:

```
symbol ex_7_2_conj : Prf(a ∨ (c ⇒ b))≔
begin
    rewrite .[x in _ ∨ ( _ ⇒ x)] simp1_eq;
    refine ex_7_axiom
end;
```

4.3 Additional Transformations

Leo-III can transform literals or clauses in various ways before or during the application of inference rules. These transformations generally maintain equivalence and can be as simple as changing the order of literals, but they need to be accounted for in verification. This will often necessitate additional inference rules, referred to as *accessory rules* in the following. These are then applied before and after the actual calculus rule, leading to additional steps in the encoding. Therefore, the implementation of each calculus rule needs to be analyzed to identify the possible implicit transformations and each transformation occurring in a concrete Leo-III proof step needs to be accounted for in the encoding.

4.3.1 Transforming Literals to Their Equational and Non-Equational Forms

As we have seen, the inference rules of Leo-III commonly operate on equality literals. Non-equational literals are however represented as $[p_o]^{tt}$ rather than in the equivalent equational form $[p_o \simeq \top]^{tt}$. While it would be possible to provide several alternative encodings for inference rules to enable them to also take non-equational literals as premises, this would necessitate a number of alternative encodings to account for the different possible combinations of equational and non-equational literals as premises. Encoding the inference rules for equational literals and transforming non-equational ones to their equational format before the application of inference rules, and conversely transforming them back afterwards, therefore represents the more elegant alternative. Many rules also require the literals they operate on to have a specific polarity. This is not a problem, since non-equational literals of either polarity can easily be transformed to equational ones of arbitrary polarity through the appropriate negations. This versatility can be accounted for through the use of four separate accessory rules that transform non-equational to equational literals:

$$\frac{[x_o]^{ff}}{[x_o \simeq \top]^{ff}}\ \text{NegPropNegEq} \qquad \frac{[x_o]^{ff}}{[\neg x_o \simeq \top]^{tt}}\ \text{NegPropPosEq}$$

$$\frac{[x_o]^{tt}}{[\neg x_o \simeq \top]^{ff}}\ \text{PosPropNegEq} \qquad \frac{[x_o]^{tt}}{[x_o \simeq \top]^{tt}}\ \text{PosPropPosEq}$$

The rules for the transformation back to non-equational literals are the exact counterparts:

$$\frac{[x_o \simeq \top]^{ff}}{[x_o]^{ff}}\ \text{NegEqNegProp} \qquad \frac{[\neg x_o \simeq \top]^{tt}}{[x_o]^{ff}}\ \text{PosEqNegProp}$$

$$\frac{[\neg x_o \simeq \top]^{ff}}{[x_o]^{tt}}\ \text{NegEqPosProp} \qquad \frac{[x_o \simeq \top]^{tt}}{[x_o]^{tt}}\ \text{PosEqPosProp}$$

Note that such transformations could be avoided (except in cases where the polarity of a literal has to be adjusted) if all literals would by default be encoded as equalities. This however makes for a less readable representation that does not correspond directly to the output of Leo-III in the TSTP format. Therefore, the approach presented above was chosen.

Another similar transformation we will use in the encoding is the following:

$$\frac{[x_o]^{tt}}{[\top \simeq x_o]^{tt}}\ \text{TopEqPosProp} \qquad \frac{[x_o]^{ff}}{[\bot \simeq x_o]^{tt}}\ \text{BotEqNegProp}$$

Since these accessory rules operate on individual literals and the equational and non-equational representations are equivalent, an encoding as an equality proof is possible and will facilitate the application of

these rules in the verification of Leo-III proof steps. Furthermore, the ten rules can be encoded statically as shown in the following.

Proof-Scripts. The proofs of the rules as equalities generally follow the same pattern: Recall that `propExt` is a Lambdapi-axiom that can be instantiated with two propositions and then maps encodings of the proofs of the mutual implication of the propositions to a proof of their equality. This is utilized here by instantiating `propExt` with the non-equational and equational forms of the literal, yielding a function that will provide a proof of the equality when subproofs of the two implications are supplied.

The proof of the simple case of the encoding of PosPropPosEq is shown here, the other proofs are similar but involve handling of the negations through he use of the rules `¬I` and `¬E` and can be found in appendix B.

```
1  opaque symbol posPropPosEq_eq x: (Prf (x = ( x = ⊤))):=
2  begin
3      assume x;
4      refine propExt x (x = ⊤) _ _
5          {assume h1;
6          refine propExt x ⊤ _ _
7              {assume h2;
8              refine ⊤I}
9              {assume h2;
10             refine h1}}
11         {assume h2;
12         refine (=def [o] x ⊤ h2 (λ z, z)) ⊤I}
13 end;
```

Here, the implication `Prf x → Prf (x = ⊤)` is shown by assuming `h1: Prf x` (line 5) and then showing `x = ⊤` by once more using `propExt` (line 6) and proving both `Prf x → Prf ⊤` and `Prf ⊤ → Prf x` individually. In both cases, the premises of the implications are assumed (lines 7 and 9) leaving the obligations to provide proof-terms for `⊤` and `x` respectively. `⊤I` provides a proof of the former (line 8) and `h1` of the latter (line 10).

To show `Prf (x = ⊤) → Prf x`, `h2: Prf (x = ⊤)` is assumed (line 11), leaving `Prf x` as the focused goal. This can then be shown using an instantiation of `=def` with `x`, `⊤` and `h2`, that results in a term of type `Π p: (El o → El o), Prf (p ⊤) → Prf (p x)`. By instantiating `p` with the anonymous function decoding identity `λ z, z`, we get a function mapping `Prf ⊤` to `Prf x`. A proof of `⊤` is then provided by `⊤I` and mapped to the desired `Prf x` (line 12).

The accessory rule for the opposite transformation, PosEqPosProp can then simply be proven using the tactic `symmetry` which transforms the obligation to prove PosEqPosProp to the obligation to prove the already shown PosPropPosEq. Alternatively, interdependencies between the proofs can be avoided by simply constructing the new proof with an analogous use of `propExt`. The subproofs will then simply have to be provided in the opposite order. Since the proofs are analogous to the ones already given, they are omitted from the appendix.

Proof-Terms. The proof-terms for the transformations are exact correspondences of the proofs of the two implications given above:

```
1  symbol posPropPosEq_term : Π x: El o, Prf x → Prf (x = ⊤)
2  ≔ λ x h1, propExt x ⊤ (λ _, ⊤I) (λ _, h1);
3
4  symbol posEqPosProp_term : Π x: Prop, Prf (x = ⊤) → Prf x
5  ≔ λ x h2, =def x ⊤ h2 (λ z, z) ⊤I;
```

Due to this correspondence, the proof-terms are also omitted from the appendix.

4.3.2 Changing Order within Equality Literals

The terms on the right- and left-hand side of equality literals in Leo-III are arranged in accordance with a term ordering [62], rules that operate on literals or their substructures can thus influence their order. To account for this in Lambdapi-proofs, an accessory rule postulating the symmetry of equality is necessary. As discussed in section 4.1.2, this is expressed in the inference...

$$\frac{[x \simeq y]^\alpha}{[y \simeq x]^\alpha} =\text{-}Sym$$

The parent and child formulae of these accessory rules are equivalent and the rules operate on individual literals. When proof-scripts are used, proving the rule as an equality term and applying the rewrite tactic is therefore used while the encoding as a proof-term encodes the symmetry as a classical proof function. In this case, the rule has to be encoded with one version for positive literals and one for negative literals.

Proof-terms. A proof-term is constructed by assuming `h1: Prf(x = y)` and instantiating `=def` with `x`, `y` and `z` as well as the anonymous function `Prf(λ z, y = z)`, resulting in a term of type `Prf (y = y) → Prf (y = x)`. The application of the Lambdapi-axiom `=ref` instantiated with `y` provides the argument of this function and results in the desired term of type `Prf (y = x)`. The proof-term is therefore given by...

```
opaque symbol eqSym_p : Π T: MonoSet, Π x: El T, Π y: El T, Prf (x = y) → Prf (y
    = x) ≔
λ T x y h1, ((=def x y h1 (λ z, y = z)) (=ref y));
```

The proof of the negated version assumes `h1: Prf(¬(x = y))` and then needs to construct a term of type `Prf(¬(y = x))`. The negation is introduced using `¬I` by assuming `h2: Prf(y = x)` and mapping it to a term of type `Prf (x = y)` using the proof of `=sym_pos` as a subproof. This term and `h1` thus form a contradiction and provide a term of the type `Prf(¬(y = x))`.

```
opaque symbol eqSym_n : Π T: MonoSet, Π x: El T, Π y: El T, Prf (¬ (x = y)) →
   Prf (¬ (y = x))≔
(λ T x y h1, ¬I (y = x) (λ h2, ¬E (x = y) ((λ h3, (=def [T] y x h3) (λ z, x = z)
    (=ref [T] x)) h2) h1));
```

Proof-script. To prove the equality encodings of the accessory rule used for the proof-scripts, the Lambdapi-axiom `propExt` and proofs of both implications `Prf(x = y) → Prf(y = x)` and `Prf(y = x) → Prf(x = y)` are used. The proofs of these is simpler when using proof-scripts, since the tactic `symmetry` is available.

```
opaque symbol eqSym_eq [T] (x y : El T) : Prf((x = y) = (y = x))≔
begin
    assume T x y;
    have H1: Prf(x = y) → Prf(y = x)
        {assume h;
        symmetry;
        refine h};
    have H2: Prf(y = x) → Prf(x = y)
        {assume h;
        symmetry;
```

```
11          refine h};
12      refine propExt (x = y) (y = x) H1 H2
13  end;
```

Since negative equational literals $[s \simeq t]^{ff}$ are encoded as negated equalities $\neg(s = t)$, we can simply use the encoded equality proven above and include the negation in the rewrite patterns.

4.3.3 Literals Changing Order

The application of some Leo-III inference rules changes the order of literals in a clause. This generally occurs when a rule operates only on a specific subset of the literals. In these cases, the Leo-III implementation selects these literals, performs the transformations, and then constructs the derived clause as a disjunction of the transformed literals and the unaffected ones. Consequently, the order of literals in the resulting clause is determined by their grouping and may differ from the original order. Due to the commutativity of $\vee$, this transformation is permissible but must be explicitly proven in Lambdapi. Additionally, changing the order of literals in the Lambdapi encoding may be necessary before applying statically encoded Lambdapi rules that require literals to be in a specific order. To verify these steps, a transformation affecting the entire clause rather than a single literal is necessary. Therefore, the rule is encoded as a proof function. As discussed in section 4.1.3, this accessory rule must be flexibly adapted to specific clauses and is hence generated on-the-fly.

We first consider the simple case of a clause with only two literals and change their order:

```
1  opaque symbol permute_1_0 x0 x1: (Prf (x0 ∨ x1)) → (Prf (x1 ∨ x0)):=
2  begin
3      assume x0 x1 h0;
4    refine (∨E x0 (x1) (x1 ∨ x0) _ _ h0)
5      {assume h1;
6      refine (∨Ir x1 (x0) h1)}
7      {assume h1;
8      refine ((∨Il x1 (x0) h1))}
9  end;
```

The proof uses `∨E` to deconstruct the obligation to prove the permuted clause (`(Prf (x1 ∨ x0))`) based on the original clause (`(Prf (x0 ∨ x1))`) into proving that both the leftmost literal of the original clause (`x0` in this example) and the remaining clause (`x1`) imply the permuted clause individually. This is line 4 in the proof.

These proofs are given in lines 5-6 and 7-8, respectively. They first assume the premise of the functions they are proving as a new hypothesis `h1` (lines 5 and 7). In the case of the left-hand side, `h1` is of type `Prf x0` and can thus be used with the introduction rule `∨Ir` to construct a term of type `Prf (x1 ∨ x0)` (line 6). The right-hand side is proven analogously.

Longer clauses with arbitrary arrangements can be proven following the same pattern, though the proofs are slightly more complex. Decomposing the clauses step by step through a nested application of the elimination rule `∨E` is necessary because the sub-clauses resulting from the first application of the elimination rule will contain more than one literal and will not necessarily appear as a whole in the clause to be derived. Thus, a simple application of the introduction rule will not suffice, we must first decompose the clause until only a single literal remains. The use of the introduction rule to prove a disjunction based on a single literal also becomes more complex as the applications of $\vee Ir$ and $\vee Il$ must be adapted to the position of the literal in the new clause, again requiring nested applications in the `refine` steps. An automatically generated definition of such a rule for a more complex case is given below to illustrate this:

```
opaque symbol permute_2_3_1_0 x0 x1 x2 x3: (Prf (x0 ∨ x1 ∨ x2 ∨ x3)) → (Prf (x2
    ∨ x3 ∨ x1 ∨ x0)):=
begin
    assume x0 x1 x2 x3 h0;
  refine (∨E x0 (x1 ∨ x2 ∨ x3) (x2 ∨ x3 ∨ x1 ∨ x0) _ _ h0)
    {assume h1;
    refine (∨Ir x2 (x3 ∨ x1 ∨ x0) (∨Ir x3 (x1 ∨ x0) (∨Ir x1 (x0) h1)))}
    {assume h1;
    refine (∨E x1 (x2 ∨ x3) (x2 ∨ x3 ∨ x1 ∨ x0) _ _ h1)
      {assume h2;
      refine (∨Ir x2 (x3 ∨ x1 ∨ x0) (∨Ir x3 (x1 ∨ x0) (∨Il x1 (x0) h2)))}
      {assume h2;
      refine (∨E x2 (x3) (x2 ∨ x3 ∨ x1 ∨ x0) _ _ h2)
        {assume h3;
        refine (∨Il x2 (x3 ∨ x1 ∨ x0) h3)}
        {assume h3;
        refine (∨Ir x2 (x3 ∨ x1 ∨ x0) (∨Il x3 (x1 ∨ x0) h3))}}}
end;
```

The naming convention used here represents the one-line notation of the applied permutation as a postfix of the name `permute`.

Proof-terms. The proof-term encodings are a direct correspondence of the scripts.

```
opaque symbol permute_1_0_term : (Π x0, Π x1, ((Prf (x0 ∨ x1)) → (Prf (x1 ∨ x0))
    )):=
λ x0 x1 h0, (∨E x0 (x1) (x1 ∨ x0) (λ h1, (∨Ir x1 (x0) h1)) (λ h1, (∨Il x1 (x0)
    h1)) h0);
```

```
opaque symbol permute_2_3_1_0_term : (Π x0, Π x1, Π x2, Π x3, ((Prf (x0 ∨ x1 ∨
    x2 ∨ x3)) → (Prf (x2 ∨ x3 ∨ x1 ∨ x0)))):=
λ x0 x1 x2 x3 h0, ∨E x0 (x1 ∨ x2 ∨ x3) (x2 ∨ x3 ∨ x1 ∨ x0) (λ h1, (∨Ir x2 (x3 ∨
    x1 ∨ x0) (∨Ir x3 (x1 ∨ x0) (∨Ir x1 (x0) h1)))) (λ h1, (∨E x1 (x2 ∨ x3) (x2 ∨
     x3 ∨ x1 ∨ x0) (λ h2, (∨Ir x2 (x3 ∨ x1 ∨ x0) (∨Ir x3 (x1 ∨ x0) (∨Il x1 (x0)
    h2)))) (λ h2, (∨E x2 (x3) (x2 ∨ x3 ∨ x1 ∨ x0) (λ h3, (∨Il x2 (x3 ∨ x1 ∨ x0)
    h3)) (λ h3, (∨Ir x2 (x3 ∨ x1 ∨ x0) (∨Il x3 (x1 ∨ x0) h3))) h2)) h1)) h0;
```

4.3.4 Double Literal Deletion

If an inference rule creates a clause containing multiple occurrences of one literal - for instance when a rule operating on literals transforms two literals into an identical one - Leo-III can only add the first occurrence and omits the others. This omission has no influence on the truth value of the clause, which is determined by the presence or absence of any literals evaluating to true. Whether only one or multiple literals evaluate to true however makes no difference, and similarly, the number of literals evaluating to false is irrelevant. Therefore, multiple occurrences of the same literal are superfluous. While there is a simplification rule for $s \vee s \to s$ ($Simp1$), using this for clauses would require us to first change the order of the literals and, in the case of more than two occurrences, apply the simplification in a nested fashion. It is thus more elegant to define a new, flexible accessory rule for this operation and generate instances on-the-fly. While this is not yet implemented, the proof presented in the following can be automatically adapted analogously to the proof of the `permute` rules. This rule transforms the clause as a whole and is thus encoded as a function mapping the original clause to the simplified one both in the encoding as a proof-script and as a proof-term. As we will see, the proofs of this rule and the accessory rule proving the permutation of clauses are very similar.

Note that this principle is also encoded by one of the rules for clause simplification in the extended calculus EP of Leo-III:

$$\frac{C \vee [s \simeq t]^\alpha \vee [s \simeq t]^\alpha}{C \vee [s \simeq t]^\alpha} \text{ (DD)}$$

In this work, we will however only encounter it as an implicit transformation carried out by other inference rules. The encoding given in the following can however also be used to encode instances of the application of (DD).

Proof-scripts. Consider for instance a rule for a clause where the second and third literal are identical.

```
1  opaque symbol delete_0_1_1 : (Π x0, Π x1, ((Prf (x0 ∨ x1 ∨ x1)) → (Prf (x0 ∨ x1)
       ))):=
2  begin
3      assume x0 x1 h0;
4      refine (∨E x0 (x1 ∨ x1) (x0 ∨ x1) _ _ h0)
5          {assume h1;
6          refine (∨Il x0 x1 h1)}
7          {assume h1;
8          refine (∨E x1 x1 (x0 ∨ x1) _ _ h1)
9              {assume h2;
10             refine (∨Ir x0 x1 h2)}
11             {assume h2;
12             refine (∨Ir x0 x1 h2)}}
13 end;
```

Analogously to the proof of the permutation rule, the obligation to prove the simplified clause based on the original one is deconstructed using `∨E`. The elimination rule is instantiated with the leftmost literal (in this case `x0`) and the remaining clause (`x1 ∨ x1`) as well as the simplified clause we are trying to prove (`x0 ∨ x1`), resulting in a function that maps terms of types `Prf x0 → Prf (x0 ∨ x1)` and `Prf (x1 ∨ x1) → Prf (x0 ∨ x1)` as well as `h0 : Prf(x0 ∨ x0 ∨ x1)` to the desired proof of `(x0 ∨ x1)` (line 4). `_` is used in the positions of both of these arguments, which creates two now proof goals for them. Again, we first assume the premises (lines 5 and 7). For the first goal, that provides us with term `h1: Prf x0`, and leaves `Prf (x0 ∨ x1)` as a focused goal. The instantiation of `∨Il` (line 6) maps `h1` to a term of the desired type. Note that in these proofs, the assumed term will always provide a proof for one of the literals in the simplified clause since any literal of the original clause still occurs (exactly once) in the simplified one. The new clause can thus always be proven based on the assumption of the leftmost literal in such a fashion. The proof of the second implication then again uses `∨E` to decompose the proof goal further (line 8), once again requiring two subproofs: For one, `∨Il` has to be used to once more show that the leftmost literal of the shortened clause implies the simplified clause. The second subproof then uses `∨E` again to divide the original clause further. This can be repeated an arbitrary number of times to prove clauses of an arbitrary length. Once a clause with only two literals remains, in our example this is the case in line 7, `∨E` can be applied one more time and the two created proof goals can both be proven with `∨Il` and `∨Ir` (lines 10 and 12).

This proof can be extended to clauses of arbitrary lengths and literals making an arbitrary number of occurrences in an arbitrary order following the pattern described above.

Proof-terms. The proof-term of the above example is in exact correspondence to the proof presented as a script:

```
1  opaque symbol delete_0_1_1_term : Π x0: Prop, Π x1: Prop, Prf (x0 ∨ (x1 ∨ x1)) →
       Prf (x0 ∨ x1)
2  := λ x0 x1 h0, ∨E x0 (x1 ∨ x1) (x0 ∨ x1) (λ h1, ∨Il x0 x1 h1) (λ h1, ∨E x1 x1 (x0
       ∨ x1) (λ h2, ∨Ir x0 x1 h2) (λ h2, ∨Ir x0 x1 h2) h1) h0;
```

4.3.5 Application of other Calculus Rules

Lastly, some Leo-III implementations of calculus rules themselves involve the implicit use of other rules (for example (Simp)). These cases are not listed as seperate steps in the TSTP output. We can generally verify such nested applications by simply applying the encoding of the secondary rule within the enocoding of the primary rule. We can thus not encode a specific accessory rule for such cases but instead rely on the existing encodings of the respective inference rules.

4.4 Terms vs. Scripts

It has been demonstrated that the challenges encountered in an encoding of the Leo-III proofs can be met using both proof-terms and proof-scripts. While an encoding of proofs using terms generally leads to a more compact representation, it has several drawbacks: For one, proof-terms are less readable and can only be checked as a whole. This is a practical disadvantage in the implementation, trouble-shooting and maintenance process since it makes sources of errors harder to identify. The more significant advantage of proof-scripts is however the possibility of using additional tactics, particularly `rewrite`, in scripts. This not only dispenses with a number of accessory rules that would otherwise have to be generated on-the fly or applied in a nested fashion, but also makes the encoding more robust because the tactic is by design more easily adaptable to specific terms. Lambdapi is constantly extended with further tactics that facilitate proofs using scripts and utilizing this potential is a valuable resource in proof encodings. Since longer outputs are generally not considered to be a problem in this project, the advantages of proof-scripts therefore outweigh the longer outputs. Furthermore, a more concise formulation of the proofs in scripts than chosen in the examples here would also be possible. This could for instance be achieved by integrating subproofs into the `refine` steps as proof-terms. In the following, proof-scripts will thus be used in the modular encoding strategies of the calculus rules applied in Leo-III. Each of the approaches discussed in this chapter then becomes a step in a Lambdapi encodings of an individual EP-rule application in Leo-III proofs. For readability and to highlight the modularity of the encoding, each step will be represented using explicit subproofs, for instance by defining a hypothesis with the `have` tactic. In an implementation, much shorter encodings are however also possible.

4.5 Summary: General Approaches for Proof-Scripts

The following accessory rules are used:

Dynamically generated rules

- `transformation` rules construct functions operating on the proofs of whole clauses based on functions operating on proofs of individual literals.
- `permutation` rules prove that the order or literals can be changed arbitrarily.
- `delete` rules proof that double occourences of literals can be omitted.

Statically encoded rules

- `posPropPosEq_eq` etc. prove the conversion between equational and non-equational forms of literals.
- `eqSym_eq` proofs that the left- and right-hand sides of equational literals can be exchanged.

The following table summarizes the encoding approaches and the necessary accessory rules derived for the different general encoding demands identified for the implemented Leo-III calculus rules.

	Encoding Approach	Accessory Rules
Adaptability of rules		
Static	Encode rules as a single Lambdapi terms.	-
Versatile	Encode rules as multiple Lambdapi terms for different versions of the rule.	-
Flexible	Generate instances and proofs of the encoding tailored to the specific clause at hand on-the-fly.	-
Structure rule operates on		
Clause	Encode rules as functions.	-
Literal	Encode rules as equalities and use the `rewrite` tactic or (if conclusions of rules are only consequence but not equivalent to the premises) encode as functions and use the `transform` rules.	`transform`
Substructures of literals	Encode rules as equalities and use the `rewrite` tactic.	-
Implicit Transformations		
Changing between equational and non-equational form	Rewrite literals using equality rules.	`posPropPosEq_eq` etc.
Changing the order within equality literals	Rewrite literals using equality rules.	`eqSym_eq`
Changing the order of literals	Generate `permute` rule and apply to it to the clauses.	`permute`
Deletion of double literals	Generate `delete` rule and apply to it to the clauses.	`delete`
Application of other EP-rules	Apply the derived encoding of the respective rule	-

Figure 10: Summary of the encoding approaches.

5 Modular Encoding of the EP Core Calculus Rules

In this section, the modular encodings of the individual rule applications of Leo-III are derived based on the encoding approaches of the previous section. In each of the following subsections, we will first consider the rules themselves and present either a static, a versatile, or a flexible encoding. Only the types of the symbols representing calculus rules are stated here, the Lambdapi-Theorems and discussions of selected proofs can be found in Appendix D. The implementation of calculus rules in Leo-III is analyzed to identify which additional steps may be necessary in the verification. The approaches derived for these additional steps and the application of the encoded calculus rule are then combined in a modular, step-by-step encoding that addresses all relevant factors. Lastly, an example that necessitates the additional verification steps demonstrates the encoding. Since Leo-III uses some of the principles encoded in the simplification rules implicitly in the implementation of a number of the core-calculus rules, we will discuss the encoding of the simplification steps first.

5.1 Rules of the Extended Calculus

5.1.1 Formula Simplification

Formula simplification by the rule (Simp) in Leo-III is given by the exhaustive application of the 17 boolean identities depicted in Fig. 6 on both the left- and right-hand sides of all equational literals of a clause.

Encoding of Simp. We discussed the different levels a rule can operate on in Sec. 4.2 and used the simplification rule `simp1_eq` to demonstrate the difficulties and possible strategies involved with encoding rules operating on different possible structures. This is not a coincidence. In fact, as we have seen, the encoding of simplification rules without the use of proof-scripts involves many additional rules and their nested application. This was one of the primary motivations for choosing scripts to encode proofs in this project. Naturally, simplification rules will therefore be encoded as equality literals to enable the use of the `rewrite` tactic to verify them, following the work presented in [26].

This encoding has to be carried out for each of the identities in Fig. 6. Note that in the Lambdapi encodings, we will use the simplified term as the left-hand side and the more complex one as the right-hand side. This may seem counter-intuitive at first since it means we cannot use these rules to rewrite complex terms to simplified ones. The reason for this encoding becomes clear when recalling that the `rewrite` tactic is used to transform the focused goal rather than the type of a given term. When trying to verify a simplification step, the focused goal for which we will have to provide a proof-term will be the simplified clause. The idea is then to apply the encoded simplification rules to transform substructures of the simplified clause presented in the focused goal back to the more complicated form. After doing this for each of the applied simplifications, we will have transformed the focused goal to the clause prior to simplification, which is the parent formula. Hence, the term encoding the parent formula can be used to prove the focused goal. It is worth noting that equalities could also be encoded to perform the opposite rewrite operations and could then be used to proof functions mapping proofs of terms to proofs of their simplified version. This however makes for a more lengthy proof encoding, thus the approach described above is generally used for rules encoded as equalities in this project.

The Lambdapi-Theorems encoding all 17 simplification rules are given in Appendix C. As an example, the type of the equality representing (Simp7) is stated here:

```
1 symbol simp7_eq x: (Prf (x = (x ∨ ⊥)));
```

While such proof-terms can be encoded statically, it is often necessary to encode two versions to account for slight structural variations. This is also the case here: In addition to `simp7_eq` as encoded above, we also need a version where the order in the disjunction is reversed, i.e., `simp7_eq_rev`, encoding `(Prf (x = (⊥ ∨ x))`. The proofs for these variations are analogous and will thus be omitted from the appendix.

Implementation in Leo-III. The implementation of (Simp) first exhaustively applies the boolean identities to the left- and right-hand sides of literals. The simplified terms are then combined into their equational literals again and, in the process, are ordered according to a term ordering. If any of these applications lead to a literal of the form $[t_T \simeq t_T]^{tt}$, Leo-III identifies it with $\top$. While there is a rewrite rule encoding this transformation (Simp9), this operation is, strictly speaking, not an instance of the calculus rule (Simp), as it simplifies the literal as a whole rather than merely the left- and right-hand sides. In the given encoding, we can however also use Simp9 on literals, as they too are encoded as equality terms. If any of the simplified literals represent a solved equality constraint, the calculus rule (Bind) is applied to carry out the encoded substitution for the remaining clause. A clause containing the literals resulting from all of the described operations is constructed, and double occurrences of literals are omitted.

Modular encoding of (Simp). In the encoding of (Simp), we use both inference rules encoded as equalities and ones encoded as functions. Since the applications of the two are quite different (equalities are used to rewrite the focused goal while functions operate on proof-terms directly), we divide the encoding into two parts: First, we provide a subproof for only the reordering within literals and the simplifications themselves, both of which are proven by rewriting the corresponding substructures of the focused goal. If (Bind) was invoked, we verify it in an additional subproof taking the proof-term verifying the simplification as the parent. Likewise, if any of the literals occurred multiple times, we generate the corresponding version of the `delete` function and use it to map the proof-term of either the unification (if it took place) or the simplification to a proof-term of the contracted version of the clause. These considerations and the resulting modular encoding of (Simp) are summarized in Fig. 11.

Categorization of (Simp) Encoding Demands

Adaptability of Rules	Structure operated on	Additional Transformations
Static	Terms	Changing the order within equality literals, Deletion of double literals, Application of other calculus rules

Modular Encoding of (Simp)

1. Assume the free variables of the child clause.
2. Verify the application of the simplifications themselves in a subproof using the `have` tactic.
   - 2 a) If the order of terms within equality literals was changed, rewrite the respective literals using `eqSym_eq`.
   - 2 b) Apply the respective encoding for each of the applied simplifications using the `rewrite` tactic.
   - 2 c) Refine with the (instantiated) parent.
3. If any of the literals represents a solved unification constraint, verify the application of (Bind).
4. If any of the literals occur multiple times, verify the removal of superfluous literals using an instance of `delete`.
5. Refine with the last derived term.

Figure 11: Categorization and Modular Encoding of (Simp)

Example. For simplicity, we will consider an example that does not involve the application of unification. Assume that `b` and `c` are encoded propositions and `delete_0_1_1` is defined as discussed in Sect. 4.3.4. Based on

```
1  symbol exSimp_parent: Π a , Prf(c ∨ (a = (b ∨ b)) ∨ ((b ∨ ⊥) = a));
```

We can then proof the following simplification:

```
1  symbol exSimp_child: Π a , Prf(c ∨ (b = a));
2  begin
3      // Step 1
4      assume a;
5      // Step 2
6      have Simplification: Prf(c ∨ (b = a) ∨ (b = a))
7          // Step 2 a)
8          {rewrite .[x in (_ ∨ x ∨ _)] (eqSym_eq [o]);
9          // Step 2 b)
10         rewrite .[x in (_ ∨ (_ = x) ∨ _)] simp1_eq;
11         rewrite .[x in (_ ∨ _ ∨ (x = _))] simp7_eq;
12         // Step 2 c)
13         refine exSimp_parent a};
14     // Step 4
15     have DeleteDoubleLiterals: Prf(c ∨ (b = a))
16         {refine delete_0_1_1 c (b = a) Simplification};
17     // Step 5
18     refine DeleteDoubleLiterals;
19 end;
```

This example illustrates how the encoded rules are used to stepwise transform the simplified clause in the focused goal of the `simplification` subproof: The goal `Prf (c ∨ ((b = a) ∨ (b = a)))` is rewritten to `Prf (c ∨ ((a = b) ∨ (b = a)))` in line 8, then the first simplification encoding rewrites it to `Prf (c ∨ (a = (b ∨ b) ∨ (b = a)))` (line 10), and the second one to `Prf (c ∨ (a = (b ∨ b) ∨ (b ∨ ⊥) = a))` (line 11), at which point it corresponds to the type of `exSimp_parent`.

Note that the order in which we apply the rewriting operations is important in case the applied simplifications overlap. Since we are reconstructing the original clause by rewriting the simplified terms to their more complex counterparts, the order in which the rewriting operations have to be applied is the reverse order of the applications of the respective original simplification rules.

5.1.2 Rewriting

As discussed previously, Leo-III can use clauses consisting of only a single literal for rewriting:

$$\frac{C \vee [s \simeq t]^{\alpha} \qquad [l \simeq r]^{tt}}{C \vee [s[r\sigma]_p \simeq t]^{\alpha}}\ (\mathrm{RW})^{\dagger}$$

†: Where $s|_p \equiv l\sigma$ for some substitution σ and $l\sigma$ is bigger than $r\sigma$ with regard to a term ordering

Encoding of (RW). Encoding (RW) is straightforward, since it corresponds to the rewrite operation applied in Lambdapi. Therefore, no Lambdapi term needs to be defined to encode (RW).

Implementation in Leo-III. There are two instances in which Leo-III will attempt to use single literal clauses for rewriting: Clauses consisting only of a positive equational literal, and clauses consisting only of a single non-equational literal. In the later case, the literals $[p_o]^{tt}$ or $[p_o]^{ff}$ (where p is a proposition), are equivalent to $[p_o \simeq \top]^{tt}$ and $[p_o \simeq \bot]^{tt}$ and can thus be used to rewrite occurrences of p_o. These rewrite-clauses are identified, substitutions are potentially applied to them if they are not ground, and then the rules are used to rewrite other clauses by replacing instances of their left-hand side with their right-hand side. After rewriting, simplification of the transformed clauses is carried out.

Modular encoding of (RW). Once again, we use `rewrite` to transform the focused goal back to match the type of the parent formula, which we can then use to provide a proof. For this purpose, we need to carry out the rewriting steps in the reverse direction and thus need to provide proof-terms encoding the corresponding equalities. In the case of non-equational literals, we first need to prove the transformation into a positive equational form with either $\top$ or $\bot$ on the left-hand side. This is achieved using the accessory rules `topPosProp_eq` and `botNegProp_eq`. In the case of equational literals, we need to first flip the right- and left-hand sides of the literal in the rewrite-clause to use them to rewrite the focused goal. A corresponding proof is provided using `eqSym_eq`.

Categorization of (RW) Encoding Demands

Adaptability of Rules	Structure operated on	Additional Transformations
-	Terms	Transforming to equality literal, Application of other calculus rules

Modular Encoding of (RW)

1. Assume the free variables of the child clause.
2. Use the `have` tactic to provide a proof-term for the equality used to rewrite the focused goal. The exact form depends on the kind of clause used as a rewrite rule by Leo-III:
   - 2 a I) If the rewrite-clause is a non-equational single literal, proof the transformation to equational form using `topPosProp_eq` or `botNegProp_eq`.
   - 2 a II) If the rewrite-clause is an equational single literal, use `=symp_pos_eq` to prove the reverse rewrite rule.
   - 2 b) Refine with the rewrite-clause and - if a substitution was applied - instanciate it accordingly.
3. Use the (transformed) rewrite-clause to rewrite the focused goal.
4. If simplifications were applied, use the encoding of (Simp) to verify the transformations.
5. Refine with the (instantiated) parent.

Figure 12: Categorization and Modular Encoding of (RW)

Example. We consider examples where (Simp) is not invoked. Assume that `b` and `c` are encoded propositions and a clause is given by

```
1 symbol exRW_parent: Prf(b ∨ c);
```

We can now use a clause consisting of a non equatinal single literal to rewrite terms to $\bot$:

```
symbol exRW_rwClause1: Prf(¬ (b ∨ c));

symbol exRW_child1: Prf ⊥ ≔
begin
    // Step 1 is not necessary here

    // Step 2
    have RewriteRule : Prf(⊥ = (b ∨ c))
        // Step 2 a I)
        {rewrite botNegProp_eq;
        // Step 2 b)
        refine exRW_rwClause1};
    // Step 3)
    rewrite RewriteRule;
    // Step 4)
    refine exRW_parent
end;
```

Or use clauses consisting of a single positive equational literal:

```
symbol exRW_rwClause2: Π a, Prf((a ∨ c) = b);

symbol exRW_child2: Prf b ≔
begin
    // Step 1 is not necessary here

    // Step 2
    have RewriteRule : Prf(b = (b ∨ c))
        // Step 2 a II)
        {rewrite (eqSym_eq [o]);
        // Step 2 b)
        refine exRW_rwClause2 b};
    // Step 3)
    rewrite RewriteRule;
    // Step 4)
    refine exRW_parent
end;
```

5.2 Extensionality Rules

5.2.1 Functional Extensionality Rules

Recall the rules for functional extensionality:

$$\frac{C \vee [s_{\tau\to\nu} \simeq t_{\tau\to\nu}]^{tt}}{C \vee [s\ X_\tau \simeq t\ X_\tau]^{tt}}\ (\text{PFE})^\dagger \qquad \frac{C \vee [s_{\tau\to\nu} \simeq t_{\tau\to\nu}]^{ff}}{C \vee [s\ sk_\tau \simeq t\ sk_\tau]^{tt}}\ (\text{NFE})^\ddagger$$

†: where X_T is fresh for C, ‡: where sk_T is a Skolem term

As discussed, the handling of steps involving the introduction of skolem terms is outside of the scope of this thesis, (NFE) is therefore left for future work. Applications of (PFE) can however be verified.

Encoding of (PFE). PFE operates on individual literals by applying implicitly quantified variables to both sides of (positive) equality literals where both sides have the same function type. The rule is encoded as a function of the following type ...

```
opaque symbol PFE [T] [S] (f : (El (S ⇝ T))) (g : (El (S ⇝ T))) (x : (El S)): ((
    Prf (f = g)) → (Prf ((f x) = (g x))));
```

The Lambdapi-proof of the rule is given in Appendix D.1. Note that the rule can deduce more than one possible literal if the premise is a literal postulating equality of function types that take more than just one argument: If we for instance consider a literal $[s_{\iota\to\iota\to\iota} \equiv t_{\iota\to\iota\to\iota}]^{tt}$, Leo-III will derive both a clause with the literal $[s_{\iota\to\iota\to\iota}X_1 \equiv t_{\iota\to\iota\to\iota}X_1]^{tt}$ and one including $[s_{\iota\to\iota\to\iota}X_1X_2 \equiv t_{\iota\to\iota\to\iota}X_1X_2]^{tt}$. It is however not necessary to account for this versatility through the definition of multiple versions of the rule, since the encoding can simply be applied in a nested fashion through the use of the appropriate `transform` rules to deduce the application of one variable at a time.

Implementation in Leo-III. Leo-III detects equality literals with terms of function types and applies (PFE) and (NFE) in one step to the positive and negative literals, respectively. To achieve this, Leo-III divides the literals of a clause into groups to which either of the rules can be applied and a group of all other literals. (NBE) is then exhaustively applied to negative literals through the generation and application of new Skolem terms. As described above, (PFE) can derive more than one possible child-literal if more than one argument can be applied to the terms of the parent-literal. In such cases, each possible literal is derived, and a clause is formed for each possible combination of these literals. The terms within the newly derived equality literals are then arranged according to the term ordering. For each combination of literals transformed by (PFE), a new clause is formed, including the literals resulting from (NFE) and those that were not affected. Thus, reordering both of terms within equality literals and of literals within clauses is possible.

Modular encoding of (PFE). Based on the encoding of (PFE) itself and the necessary additional transformations, the modular encoding detailed in Fig. 13 is defined.

Example. Given the following symbols for `f` and `g` ...

```
symbol f: El (ι ⇝ ι ⇝ ι);
symbol g: El (ι ⇝ ι ⇝ ι);
```

... As well as the rule `permute_1_0` that was defined and proven as discussed in Sect. 4.3.3 and `transform_1_0` defined and proven as in Sect. 4.2.2, assume that we have the following clause as a parent:

```
symbol exPFE_parent:  Π (a : El o), Prf((f = g) ∨ a);
```

We can then use the modular encoding to prove...

```
symbol exPFE_child: Π (a : El o) (x1: El ι) (x2: El ι), Prf(a ∨ ((g x1 x2) = (f
    x1 x2)))≔
begin
    // Step 1
    assume a x1 x2;
    // Step 2
    rewrite .[x in _ ∨ x] (eqSym_eq [ι] (g x1 x2) (f x1 x2));
    // Step 3
    have PFE_1 : Prf (f = g) → Prf (f x1 = g x1)
```

Categorization of (PFE) Encoding Demands

Adaptability of Rules	Structure operated on	Additional Transformations
Static	Literals	Changing the order within equality literals, Changing the order of literals

Modular Encoding of (PFE)

1. Assume the free variables of the child clause.

   For each affected literal:

   2. If the order within the literal was changed, apply `eqSym_eq`
   3. Instantiate `PFE` and use it to define a new hypothesis.

4. With the proven hypothesis, prove the application of `PFE` to the (instantiated) parent using the appropriate accessory `transform` rule. In the case of a nested application, use multiple applications of the `transform` rule.
5. If the order of literals was changed, generate a `permute` rule and apply it to permute the literals.
6. Refine with the last proven step.

Figure 13: Categorization and Modular Encoding of (PFE)

```
        {refine (PFE f g x1)};
    have PFE_2 : Prf (f x1 = g x1) → Prf (f x1 x2 = g x1 x2)
        {refine (PFE (f x1) (g x1) x2)};
    // Step 4
    have PFE_1_application : Prf((f x1 = g x1) ∨ a)
        {refine transform_1_0 (f = g) a (f x1 = g x1) PFE_1 (exPFE_parent a)};
    have PFE_2_application : Prf((f x1 x2 = g x1 x2) ∨ a)
        {refine transform_1_0 (f x1 = g x1) a (f x1 x2 = g x1 x2) PFE_2
            PFE_1_application};
    // Steps 5 and 6
    refine permute_1_0 (f x1 x2 = g x1 x2) a PFE_2_application;
end;
```

5.2.2 Boolean Extensionality

The boolean extensionality rules were defined by:

$$\frac{C \vee [s_o \simeq t_o]^{tt}}{\begin{array}{l} C \vee [s_o]^{tt} \vee [t_o]^{ff} \\ C \vee [s_o]^{ff} \vee [t_o]^{tt} \end{array}}\ (\text{PBE}) \qquad \frac{C \vee [s_o \simeq t_o]^{ff}}{\begin{array}{l} C \vee [s_o]^{tt} \vee [t_o]^{tt} \\ C \vee [s_o]^{ff} \vee [t_o]^{ff} \end{array}}\ (\text{NBE})$$

Like in the case of (PFE), the rules for boolean extensionality do not need to be generated on-the-fly,

but can be encoded once and for all. Each of the two has two possible conclusions, which is reflected by the encoding of each calculus rule using two Lambdapi versions. Furthermore, each of the conclusions here is a logical consequence of the premise, but not vice versa. Therefore, these rules cannot be encoded as equalities and used to rewrite the literals they operate on but are instead represented as functions.

Encoding of (PBE). Bear in mind that a disjunction of the form $\neg p \vee q$ is equivalent to $p \Rightarrow q$. It then becomes clear that creating two new clauses for the equality literal $[s_o \simeq t_o]^{tt}$, one of them containing the literals $[s_o]^{tt}$ and $[t_o]^{ff}$, the other one containing the literals $[s_o]^{ff}$ and $[t_o]^{tt}$, encodes the mutual implication of s and t. This is encoded in the following terms:

```
symbol PBE_r x y: Prf(x = y) → Prf(x ∨ (¬ y));
```

```
symbol PBE_l x y: Prf(x = y) → Prf((¬ x) ∨ y);
```

Encoding of (NBE). Proving the negative counterparts is a little more involved. Here, the two conclusions encode that, given the literal $[s_o \simeq t_o]^{ff}$ in the premise, we can conclude that either s_o or t_o is true and - simultaneously - one of them is false. This results in two new clauses, one of which contains the literals $[s_o]^{ff}$ and $[t_o]^{ff}$, the other one the counterparts $[s_o]^{tt}$ and $[t_o]^{tt}$. We first consider the encoding of the latter conclusion:

```
symbol NBE_p x y: Prf(¬(x = y)) → Prf(x ∨ y);
```

The second rule is encoded as:

```
symbol NBE_n x y: Prf(¬(x = y)) → Prf(¬ x ∨ ¬ y);
```

The proofs of the four encoded rules are stated and discussed in Appendix D.2.

Implementation in Leo-III. As in the case of functional extensionality, Leo-III implements (PBE) and (NBE) in one step and first divides the literals into those to which the rules can be applied and the rest. Each rule application to the individual literals then provides two sets of new literals and, as before, Leo-III forms a new resulting clause for each possible combination of literals. If we, for instance, run the process on a clause with two literals, to one of which we can apply (PBE) and to the other (NBE), Leo-III would derive two sets of literals from each of them and then generate four resulting clauses to account for the different possible combinations. The order of literals in the derived clauses then depends on whether or not the literals were affected and does not necessarily correspond to the original order. In cases where the rule applications generate a literal already present in the clause, or ones where the same literal results from two rule applications, only one occurrence is kept.

Modular encoding for (PBE) and (NBE). The considerations given above result in the categorization and, based on it, the modular encoding using the appropriate accessory rules, as summarized in Fig. 14.

Example. Assume that `a` and `b` encode propositions and an encoded clause is given by...

```
symbol exBE_parent: Π x1, Prf(x1 ∨ (¬ ((¬ a) = b)) ∨ (a = b));
```

The derivation of one of the four possible clauses that can result form applying (NBE) to `¬ ((¬ a) = b)` and (PBE) to `a = b` is encoded by:

Categorization of (PBE) and (NBE) Encoding Demands

Adaptability of Rules	Structure operated on	Additional Transformations
Versatile	Literals	Changing the order of literals, Deletion of double literals

Modular Encoding of (PBE) and (NBE)

1. Assume the free variables of the child clause.

   For each affected literal:

   2. Choose the correct encoding (`PBE_l` , `PBE_r` , `NBE_p` or `NBE_n`), instantiate it and use it to define a new hypothesis.

3. Proof the application of the rules using the necessary accessory `transform` rule.
4. If the rules resulted in double occurrences of literals, proof the removal using the necessary `delete` rule.
5. If the order of literals was changed, generate a `transform` rule and apply it to permute the literals.
6. Refine with the last proven step.

Figure 14: Categorization and Modular Encoding of (PBE) and (NBE)

```
symbol exBE_child : Π x1, Prf((¬ a ∨ b) ∨ x1):=
begin
    // Step 1
    assume x1;
    // Step 2
    have boolExt1: Prf (¬ ((¬ a) = b)) → Prf (¬ a ∨ b)
        {refine NBE_p (¬ a) b};
    have boolExt2: Prf (a = b) → Prf (¬ a ∨ b)
        {refine PBE_l a b};
    // Step 3
    have boolExtClause : Prf (x1 ∨ ((¬ a ∨ b) ∨ (¬ a ∨ b)))
        {refine (transform_0_1_1 x1 (¬ ((¬ a) = b)) (¬ a ∨ b) (a = b) (¬ a ∨ b)
            boolExt1 boolExt2) (exBE_parent x1)};
    // Step 4
    have deleteClauses : Prf (x1 ∨ (¬ a ∨ b))
        {refine delete_0_1_1 x1 (¬ a ∨ b) boolExtClause};
    // Steps 5 and 6
    refine permute_1_0 x1 (¬ a ∨ b) deleteClauses
end;
```

Where `transform_0_1_1` and `permute_1_0` are encoded as discussed previously.

5.3 Equal Factoring

As we have seen, equal factoring affects clauses containing two literals that can be unified. The inference rule allows the removal of one of the literals and instead adds two unification constraints that require the unifiability of both the left- and right-hand sides of the equality literals in the original clause:

$$\frac{C \vee [s_\tau \simeq t_\tau]^\alpha \vee [u_\tau \simeq v_\tau]^\alpha}{C \vee [s_\tau \simeq t_\tau]^\alpha \vee [s_\tau \simeq u_\tau]^{ff} \vee [t_\tau \simeq v_\tau]^{ff}}\ (\text{EqFact})$$

Encoding of (EqFact). Since the conclusions are merely a consequence of the premises of this rule rather than being equivalent, we cannot encode it as an equality to be used with the `rewrite` tactic. Rather, we encode it as a function mapping the two literals we are attempting to unify to the three literals representing the more general one of the original literals, as well as the two unification constraints. We will encode this rule statically and in two versions: one operating on positive and one on negative literals. In the case of clauses containing more than just the two equality literals, verifying (EqFact) will thus require us to first prove that we can move the literals to the last positions of the clause (using an appropriately generated `permute` function) and then use a `transform` function to prove the transformation of the whole clause. An alternative approach requiring fewer applications of accessory rules would be to not encode the versions of the (EqFact) rule statically but instead adapt them to the structure of the clause at hand. However, the more complex nature of the proof of the encoded rule would make this a more involved approach. Encoding the rules once and for all and adapting the clauses at hand to it is thus the more practical and robust solution.

Equal factoring for positive equality literals is represented by a term of type...

```
1 symbol EqFact_p [T] x y z v: ((Prf ((x = y) ∨ (z = v))) → (Prf ((x = y) ∨ (¬ (x
    = z)) ∨ (¬ (y = v)))));
```

The proof is stated and discussed in Appendix D.3. The corresponding version for negative literals is encoded as...

```
1 symbol EqFact_n [T] x y z v: ((Prf ((¬ ( x = y)) ∨ (¬ ( z = v)))) → (Prf ((¬ ( x
     = y)) ∨ (¬ ( x = z)) ∨ (¬ ( y = v)))));
```

Implementation in Leo-III. To determine which combinations of literals of a clause are suitable candidates for the application of the (EqFact) rule, Leo-III first identifies the most general literals through a comparison with regard to a term ordering. Each of them is then compared to each of the remaining literals to determine whether equal factoring can be applied to them. If both literals have the same polarity, Leo-III forms a pair of their left- and right-hand sides for each of the two possible combinations. The implementation narrows down the selection of literals to which the rule is applied by checking for criteria that would either ensure that the terms would be unifiable or prohibit it. The rule is then applied to the literals that are deemed to be potentially unifiable. If the literals do not share the same polarity, equal factoring can still be applied in cases where at least one of the literals is non-equational with a variable as a head symbol, as in such cases the literals can be transformed into equational literals of either polarity. These literals are thus also included in the analysis. For each of the pairs of literals that have passed this test, the actual equal factoring is applied, and for each of these different combinations, a separate resulting clause will be generated. The actual factoring is then implemented by forming the unification constraints, ordering the equality literals with regard to the term ordering, and constructing the new clauses. In this process, (Simp) is applied to all literals other than the unification constraints.

Modular encoding of (EqFact). These considerations give rise to the classification and modular encoding summarized in Fig. 15

Categorization of (EqFact) Encoding Demands

Adaptability of Rules	Structure operated on	Additional Transformations
Versatile	Literals	Transforming to equality literal, Changing the order within equality literals, Changing the order of literals

Modular Encoding of (EqFact)

1. Assume the free variables of the child clause.
2. Identify the two literals to be unified and compose a function proving the rule application including all necessary transformations:
   - 2 a) If the order of the left- and right-hand sides in either of the literals has to changed in order for the encoded equal factoring rule to associate the sides correctly, apply `eqSym_eq`.
   - 2 b) If either of the literals is not equational, transform to the equational form with the correct polarity.
   - 2 c) Apply the appropriate version of equal factoring (`EqFact_p` or `EqFact_n`).
   - 2 d) If the order within any of the equality literals has changed after the rule application as a result of the term ordering, proof the transformation using `eqSym_eq`.
   - 2 e) Prove the transformation to non-equational literals of any literals that are non-equational.
3. If the clause has more than two literals and the order of the literals does not match the one required by the derived function, i.e. if the literals to be unified are not at the last positions of the clause, proof the permuted clause using an instance of `permute`.
4. Apply the rule to the two literals to be unified using the appropriate `transform` function.
5. If the order of literals was changed, change it back with the correct instance of `permute`.
6. Refine with the last proven step.

Figure 15: Categorization and Modular Encoding of (EqFact)

Example. The following example is given without the application of (Simp) on the derived clause.

Let `a`, `b` and `c` be propositions and `permute_1_0_2`, `permute_1_2_0_3` and `transform_0_1` accessory rules defined and proven as discussed in the respective sections. We then consider...

```
symbol exEqFact_parent : Π x0, Prf((⊤ = x0) ∨ c ∨ (¬ b));
```

And proof the application of (EqFact):

```
symbol exEqFact_child: Π x0, Prf(x0 ∨ ¬((¬ b) = x0) ∨ c  ∨ (¬ ⊤)):=
begin
    assume x0;
    have equalFactoring: Prf((⊤ = x0) ∨ (¬ b)) → Prf(x0 ∨ ¬((¬ b) = x0) ∨ (¬ ⊤))
        // Step 1
        {assume h1;
        // Step 2
        have SymAndTransformLiteralsToEquationalForm : Prf((x0 = ⊤) ∨ ((¬ b) = ⊤
            ))
            {// Step 2 a)
            rewrite .[x in (x ∨ _)] (eqSym_eq [o]);
            // Step 2 b)
            rewrite .[x in (_ ∨ x)] (posEqNegProp_eq);
            refine h1};
        have EqualFactoringStep: Prf((x0 = ⊤) ∨ ¬(x0 = (¬ b)) ∨ ¬(⊤ = ⊤))
            // Step 2 c)
            {refine EqFact_p [(o)] x0 ⊤ (¬ b) ⊤
                SymAndTransformLiteralsToEquationalForm};
        have SymAndTransformLiteralsBackFromEquationalForm: Prf(x0 ∨ ¬((¬ b) =
            x0) ∨ (¬ ⊤))
            {// Step 2 d)
            rewrite .[x in (_ ∨ ¬ x ∨ _)] (eqSym_eq [o]);
            // Step 2 e)
            rewrite .[x in (x ∨ _ ∨ _)] posPropPosEq_eq;
            rewrite .[x in (_ ∨ _ ∨ x)] negPropNegEq_eq;
            refine EqualFactoringStep}};
        refine SymAndTransformLiteralsBackFromEquationalForm;
    // Step 3
    have Permutation: Prf(c ∨ (⊤ = x0) ∨ (¬ b))
        {refine permute_1_0_2 (⊤ = x0) c (¬ b) (exEqFact_parent x0)};
    // Step 4
    have ApplyEqualFactoringRule: Prf(c ∨ x0 ∨ ¬((¬ b) = x0) ∨ (¬ ⊤))
        {refine transform_0_1 c ((⊤ = x0) ∨ (¬ b)) (x0 ∨ ¬((¬ b) = x0) ∨ (¬ ⊤))
            equalFactoring Permutation};
    // Step 5
    have Permutation2: Prf(x0 ∨ ¬((¬ b) = x0) ∨ c  ∨ (¬ ⊤))
        {refine permute_1_2_0_3 c x0 (¬((¬ b) = x0)) (¬ ⊤)
            ApplyEqualFactoringRule};
    // Step 6
    refine Permutation2
end;
```

5.4 Unification Rules

5.4.1 Triv

Clauses containing the self evidently false literal $[s_T \simeq s_T]^{ff}$ can be omitted from the clause since, in such cases, the truth value of the clause depends entirely upon the other literals. This is encoded by (Triv):

$$\frac{C \vee [s_T \simeq s_T]^{ff}}{C}\ (\text{Triv})$$

Encoding in Lambdapi. The encoding of (Triv) can be seen as the subsequent application of two of the simplification rules discussed in Sec. 5.1.1: (Simp10) identifies terms $s_T \neq s_T$ with with $\bot$ and (Simp7) allows us to remove $\bot$ from a disjunction. Since both an arbitrary term $s_T \neq s_T$ and an equality literal $[s_T \simeq s_T]^{ff}$ are encoded as `¬(s = s)`, we can simply use the (statically) encoded simplification rules in the fashion as discussed in Sect. 5.1.1 to rewrite the focused goal in a reverse fashion. This will result in a goal corresponding to the parent formula, which we can then use to prove the goal. No additional accessory rules are therefore necessary.

Implementation in Leo-III. (Triv) is invoked as a part of the implementation of the other unification rules: Whenever the application of inference rules results in unification constraints $[s_T \simeq s_T]^{ff}$, Leo-III simply does not add it to the resulting clause.

Modular encoding of (Triv). The encoding demands and the steps of the encoding are given in Fig. 16.

Categorization of (Triv) Encoding Demands

Adaptability of Rules	Structure operated on	Additional Transformations
Static	Literals	-

Modular Encoding of (Triv)

1. Rewrite the focused goal with `simp7_eq` (or `simp7_eq_rev`).
2. Rewrite with the (instanciated) `simp10_eq`.

Figure 16: Categorization and Modular Encoding of Triv

In contrast to the other rules discussed so far, (Triv) will not occur on its own but rather be included in the verification of other inference rules. Therefore, we do not need to explicitly assume the free variables of the child formula, or refine with the parent at the end, as these tasks are left to the steps taken in the verification of primary rule that invoked (Triv). For this reason, we will not state an example of (Triv) on its own, but rather demonstrate its use in the example given for (Bind) in the following.

5.4.2 Bind

(Bind) is the central rule in the unification procedure of Leo-III. It allows us to apply substitutions encoded by solved unification constraints and then contracts the clause by removing the constraints:

$$\frac{C \vee [x_T \simeq s_T]^{ff}}{C\{s/x\}}\ (\text{Bind})^{\dagger}$$

†: where $x_T \notin \text{fv}(s)$

Encoding of Bind. Two different operations need to be encoded to verify applications of (Bind): The substitution encoded by the solved unification constraint and the removal of the corresponding literal. As any clause with free variables is encoded as a dependent type, substituting one of the variables corresponds to an instantiation of the dependent type through the application of the respective encoded term. As in the case of (RW), the operation performed by (Bind) corresponds to an operation that can be directly carried out in Lambdapi (instantiation) and is thus not represented by a term in our Lambdapi encoding. The instantiation then affects the entire encoded clause, including the encoded unification constraint. An equational unification constraint $[x_T \simeq s_T]^{ff}$ thus becomes $[s_T \simeq s_T]^{ff}$ due to the substitution it encoded. As discussed, its removal can then be handled by the encoding of (Triv).

Implementation of (Bind) in Leo-III. In the implementation of (Bind), Leo-III identifies the solved unification constraints and replaces instances of the left-hand side in the remaining clause with the right-hand side. (Bind) is directly invoked by some other inference rules that generate unification constraints and then perform specialized operations. We have already considered the implementation of one of these rules here: (EqFact). In this case, (Bind) is tasked with carrying out the substitutions for the encoded unification constraints. Recall that, as seen in the example given for the encoding of (EqFact), the unification constraint generated by this rule can take the form $[\top \simeq \top]^{ff}$, encoded as `¬⊤`. This too represents a literal equivalent to $\bot$, and we can thus proceed similarly as in the encoding of (Triv) and use a simplification rule, in this case (Simp16), that identifies the term with $\bot$ and then verify its removal due to the encoding of (Simp7). The list of such possible additional operations may have to be extended for the implemented process following other inference rules that were not considered so far.

Modular encoding of (Bind). Fig. 17 summarizes the encoding demands and the derived modular encoding.

Categorization of (Bind) Encoding Demands

Adaptability of Rules	Structure operated on	Additional Transformations
-	Clauses	Application of other calculus rules

Modular Encoding of (Bind)

1. Define a proof-term for the substitution by instantiating the parent.

   For each unification constraint:

   2 I) If the unification constrain is equational, apply the encoding of (Triv)

   2 II) If the unification constraint in non-equational, rewrite the focused goal with `simp7_eq` (or `simp7_eq_rev`) and `simp16_eq`.

3. Refine with the proof-term of the substitution.

Figure 17: Categorization and Modular Encoding of (Bind)

Example. The following example shows the verification of (Bind) after the application of (EqFact) of the previous example.

```
symbol exUni_child: Prf((¬ b) ∨ c):=
begin
    // Step 1
    have Substitution: Prf((¬ b) ∨ ¬((¬ b) = (¬ b)) ∨ c  ∨ (¬ ⊤))
        {refine exEqFact_child (¬ b)};
    // Step 2 case I)
    // (Triv) Step 1
    rewrite .[x in _ ∨ x] simp7_eq_rev;
    // (Triv) Step 2
    rewrite .[x in _ ∨ x ∨ _] (simp10_eq o (¬ b));

    // Step 2 case II)
    rewrite .[x in _ ∨ _ ∨ x] simp7_eq;
    rewrite .[x in _ ∨ _ ∨ _ ∨ x] simp16_eq;

    // Step 3
    refine Substitution;
end;
```

6 Application Example

The encoding presented here was derived as the basis for automated generation of verifiable Leo-III proofs in the Dedukti framework. A partial automation of the encoding process was carried out to assess its suitability for this purpose. As demonstrated, a multitude of factors must be considered in encoding each rule, making a complete implementation of the encoding an extensive task that was not feasible within the scope of this thesis. Therefore, it was necessary to limit the implementation to a subset of the encoding. Rather than focusing solely on a smaller subset of rules and providing more extensive automation for their verification, the implementation was tailored to cover the rules in specific versions required to automatically produce a proof for a particular well-known benchmark problem for theorem provers: Cantor's theorem. This selective approach resulted in an implementation that is not broadly applicable to other problems at this stage. However, it allowed for a detailed assessment of the automation process in the context of a specific and complex example. Therefore, it provided valuable insights into the practical usability and effectiveness of the theoretical encoding in automation and contributed to the considerations that shaped the encoding in its current form.

The implementation was carried out as an extension to the existing Leo-III software, available at[8], was likewise written in the programming language Scala [54] and is available in its current version at [70].

6.1 Cantor's Theorem

We will verify the proof Leo-III found for Cantor's Theorem [24], which states that no function mapping from a set to its powerset is surjective. In the TPTP problem (`sur_cantor.p`), this is encoded as follows:

```
thf(sur_cantor, conjecture, (~ ( ? [F: $i > ($i > $o)] : (
                                ! [Y: $i > $o] :
                                 ? [X: $i] : (
                                   (F @ X) = Y))))).
```

TPTP problems (and proofs) are encoded as so-called annotated formulae [69], which specify the used dialect (in this case the HOL language `thf`), name the formula (`sur_cantor`), and classify the formula by assigning it a role (in this case `conjecture`), before providing the actual formula in TPTP syntax. As we will see in the proof certificates, such formulae can also contain annotations to provide additional information.

The problem here consists solely of the conjecture, which states that there exists no function `F` of type $\iota \to (\iota \to o)$ such that for all functions `Y` of type $\iota \to o$, there exists an `X` of type ι satisfying `F X = Y`. This formulation uses predicates to encode sets: `Y` is a predicate encoding a set within the powerset by mapping any element in the set to `true` and any element not in the set to `false`. Therefore, the encoding postulates that there exists no function mapping individuals to sets of individuals such that for any set of individuals `Y`, there exists an individual `X` that is mapped to `Y` by `F`. This corresponds to the statement of the theorem.

6.2 Implementation

The encoding of problems. Leo-III produces output in the TSTP format (the proof of the Cantor surjectivity problem is given in Appendix E) that reconstructs the inferences leading to the empty clause and thus the proof of the problem. This makes it possible to determine which axioms, definitions, and declarations from the given reasoning problem were actually used in the found proof. It also includes the necessary declarations of the problem and the ones of new terms introduced during the proof, as illustrated in this example with Skolem terms. All these declarations and formulas must be encoded and declared in the Lambdapi proof. This is achieved through straightforward automation of the encoding discussed in Sect. 3.

[8] https://github.com/leoprover/Leo-III

The encoding of proofs. Leo-III saves information concerning the applied inference rules and the resulting clauses in order to construct a TSTP proof. This also serves as the basis for the Lambdapi proof encoding. The proof in Lambdapi follows the patterns demonstrated in Sect. 5, encoding each rule application as an individual Lambdapi term that proves the respective step. In the proof, each term representing a proof step can then be used analogously to the parent formulae in the previous examples. The steps of the Lambdapi proof follow the TSTP proof steps precisely, providing an encoding for each intermediate clause represented by an annotated formula. There is only one exception: Some steps in the TSTP output change the representation of literals, such as transforming non-equational literals like $[\neg p_o]^{tt}$ to a form that omits the negation by adjusting the literal's polarity to $[p_o]^{ff}$. Since in these cases both Lambdapi encodings would be identical (`¬p`), these steps do not need to be verified and are filtered out before the proof encoding begins. While the TSTP proof output provides formulae representing intermediate steps and identifies the calculus rules used to derive them - sometimes including additional information for inferences like unification - it generally lacks the detail needed to fully verify the steps. This is particularly true for rules with multiple versions and implicit transformations, on which the TSTP output offers no information. Consequently, modifying the existing code to track more detailed information is necessary. The specific requirements vary from rule to rule but generally include details on the position of the literals or substructures transformed by the rules and whether specific implicit transformations were performed. With this additional information, the modular encoding proved to be a very effective strategy for verifying Leo-III proofs. As discussed, clausification was not addressed in this project. Therefore, steps requiring verification of clausification are currently declared as Lambdapi axioms without proof. Similar to the TSTP proof, the final step in the verification is a term of type `Prf ⊥` , representing the derived contradiction from the axioms and the negated conjecture.

PolaritySwitch. There is one more transformation that Leo-III applies as an initial step if the negation of the conjecture leads to a double negation. In such cases, Leo-III invokes the so-called `PolaritySwitch`, which removes the double negation in the encoding and can thus effectively be proven using the principle of double negation elimination. We have already encoded this exact transformation in a simplification rule, namely `simp17` , which allows us to rewrite a subterm of the focused goal to the equivalent version without double negation. Therefore, we can simply instantiate `simp17` and use it to prove the double negation elimination of the negated conjecture, just as we would use any other simplification rule. We will see this in the proof of Cantor's Theorem given below.

Output Files. The complete system output, both in the TSTP format and as a Lambdapi version, is given in the appendices (E and F respectively). As previously discussed, a verifiable Lambdapi proof includes not only the problem and proof encodings but also all definitions of the Lambdapi theory representing extensional type theory, all definitions of the natural deduction rules, correctness proofs, and encodings of the calculus and accessory rules. The implementation thus determines which of these encodings are necessary to verify the proof at hand and includes them following the file structure shown in Fig. 9. The first line of the Lambdapi file containing the actual encoded proof then imports all of these definitions and declarations. Since all the necessary encodings were already discussed in their respective sections, the contents of these files are not provided here again.

6.3 System Output

In the following, the initial steps of the automatically produced Lambdapi proof will be discussed to illustrate the automated encoding and demonstrate the relation to the TSTP output. We will see that even though the Lambdapi encoding implementation does not directly encode the TSTP proof, there is a close correspondence between the two as they are based upon the same information.

As mentioned, both proof encodings begin by stating all used axioms and declaring terms and types. Since the reasoning problem at hand includes neither axioms nor any user-defined types or terms, the only declarations needed are those of the Skolem terms introduced during the clausification carried out

by Leo-III. In the TSTP certificate, these declarations are encoded using annotated HOL (`thf`) formulas which are classified as type declarations using the role `type`:

```
thf(sk1_type,type,
    sk1: $i > $i > $o ).
thf(sk2_type,type,
    sk2: ( $i > $o ) > $i ).
```

The Lambdapi encoding analogously declares the Skolem terms:

```
symbol sk1: (El (ι ⇝ (ι ⇝ o)));
symbol sk2: (El ((ι ⇝ o) ⇝ ι));
```

This is followed by the conjecture:

```
thf(1,conjecture,
~ ? [A: $i > $i > $o] :
  ! [B: $i > $o] :
  ? [C: $i] :
    ( ( A @ C )
    = B ),
file('sur_cantor.p',sur_cantor) ).

thf(2,negated_conjecture,
~ ~ ? [A: $i > $i > $o] :
    ! [B: $i > $o] :
    ? [C: $i] :
      ( ( A @ C )
      = B ),
inference(neg_conjecture,[status(cth)],[1]) ).
```

In the current state of the implementation, the Lambdapi output directly states the negated conjecture:

```
symbol negatedConjecture0: ((Prf ((¬ (¬ (∃(λ (A : (El (ι ⇝ (ι ⇝ o)))), (∀(λ (B :
    (El (ι ⇝ o))), (∃(λ (C : (El ι)), ((A C) = B)))))))))));
```

The next formula in the TSTP output represents one of the inferences that changes the internal representation of formulas in clause-form but affects neither the TSTP nor the Lambdapi encoding. This is evident here, since the encoded clause is identical to the one of the previous step:

```
thf(3,plain,
    ~ ~ ? [A: $i > $i > $o] :
        ! [B: $i > $o] :
        ? [C: $i] :
          ( ( A @ C )
          = B ),
    inference(defexp_and_simp_and_etaexpand,[status(thm)],[2]) ).
```

This step is hence omitted in the Lambdapi encoding. The next operation performed is the polarity switch discussed above:

```
thf(4,plain,
    ? [A: $i > $i > $o] :
    ! [B: $i > $o] :
    ? [C: $i] :
      ( ( A @ C )
      = B ),
    inference(polarity_switch,[status(thm)],[3]) ).
```

This represents the first step that we actually want to provide a proof for in the Lambdapi encoding. The annotations of the formula states the information saved by Leo-III to enable the reconstruction of the proof: The first element of the annotation `inference` identifies `polarity_switch` as the used inference rule and the last element, `[3]` provides the number of the parent formula. The Lambdapi proof hence proves `polarity_switch` as discussed earlier and - since we omitted the clause represented by formula `[3]` in the TSTP output - directly uses the `negatedConjecture0` as a parent:

```
1  // PolaritySwitch
2  opaque symbol step4: ((Prf ((∃(λ (A : (El (ι ⇝ (ι ⇝ o)))), (∀(λ (B : (El (ι ⇝ o)
       )), (∃(λ (C : (El ι)), ((A C) = B))))))))))) ≔
3  begin
4    have PolaritySwitch0 : (Prf ((∃(λ (A : (El (ι ⇝ (ι ⇝ o)))), (∀(λ (B : (El (ι ⇝
         o))), (∃(λ (C : (El ι)), ((A C) = B)))))))) = (¬ (¬ (∃(λ (A : (El (ι ⇝ (ι
       ⇝ o)))), (∀(λ (B : (El (ι ⇝ o))), (∃(λ (C : (El ι)), ((A C) = B)))))))))))
5      {refine (simp17_eq (∃(λ (A : (El (ι ⇝ (ι ⇝ o)))), (∀(λ (B : (El (ι ⇝ o))), (
          ∃(λ (C : (El ι)), ((A C) = B)))))))};
6    rewrite PolaritySwitch0;
7    refine (negatedConjecture0)
8  end;
```

The terms representing the application of inference rules are named after their respective steps in the TSTP encoding, in this case resulting in `step4`. The names of the inference rules given as annotations in the TSTP proofs are indicated by a comment preceding the clause.

The remaining inference rule applications, for instance of functional extensionality

```
thf(7,plain,
    ! [B: $i,A: $i > $o] :
      ( ( sk1 @ ( sk2 @ A ) @ B )
      = ( A @ B ) ),
    inference(func_ext,[status(esa)],[6]) ).
```

Are then verified exactly as demonstrated in the examples of the respective inference rules in Sec. 5:

```
1  // FuncExt
2  opaque symbol step7: (Π (B : (El ι)), Π (A : (El (ι ⇝ o))), (Prf (((sk1 (sk2 A)
       B) = (A B))))) ≔
3  begin
4    assume B A;
5    have PFE_0 : ((Prf ((sk1 (sk2 A)) = A)) → (Prf ((sk1 (sk2 A) B) = (A B))))
6      {refine (PFE (sk1 (sk2 A)) A B)};
7    have FunExtApplication : (Prf (((sk1 (sk2 A) B) = (A B))))
8      {refine (PFE_0 (step5 A))};
9    refine (FunExtApplication)
10 end;
```

Until, finally, the contradiction is derived and the proof is complete.

```
thf(373,plain,
    $false,
    inference(simp,[status(thm)],[372]) ).
```

```
// RewriteSimp
opaque symbol step373: ((Prf (⊥))) ≔
begin
  have TransformToEqLits : (Prf (⊥ = (sk1 (sk2 (λ (A : (El ι)), (¬ (sk1 A A))))
      (sk2 (λ (A : (El ι)), (¬ (sk1 A A)))))))
    {rewrite botNegProp_eq;
    refine (step33)};
  rewrite TransformToEqLits;
  refine (step265)
end;
```

7 Conclusion and Outlook

7.1 Conclusion

In this thesis, the potential requirements of proof encodings in the context of more complex HOL ATPs were assessed, and general encoding strategies addressing them were developed and compared, both as proof-terms and proof-scripts. The tactics available in the latter proved to be a valuable asset, and proof-scripts were hence used throughout the remainder of the thesis. Based on the general encoding approaches, a modular encoding was derived to verify the application of individual inference rules by the prover. The strategies employed depended mainly on three factors: Firstly, rules vary in the flexibility their encoding must accommodate. For instance, multiple versions of an inference rule may need to be encoded, or rules may require significant structural adaptation and thus need to be generated and proven on a case-by-case basis. Secondly, many rules operate on substructures of clauses, but rules encoded as functions only accept terms encoding entire clauses as arguments. While it is feasible to encode such rules as functions when combining them with a complex nested application of additional rules, the `rewrite` tactic available in proof-scripts significantly simplifies verification by allowing the rewriting of specific parts of the formulas. Thirdly, implicit transformations applied by the implementation of Leo-III, which are not defined by the inference rules but are consequences of the implementation, can alter clauses. These transformations must be incorporated into proofs by introducing accessory rules in additional steps. The proof of Cantor's Theorem served as a benchmark problem to demonstrate that the derived theoretical encoding can be effectively implemented.

The encodings developed here establish general theoretical foundations for HOL ATP verification within the Dedukti framework and lay the groundwork for implementing a Lambdapi extension of Leo-III. This extension will make Leo-III the first HOL ATP system to directly output Dedukti/Lambdapi certificates, paving the way for the verification of application domains that the currently available ATPs with verifiable output simply lack the expressiveness to encode. Furthermore, Leo-III will also directly bring together non-classical logics and Dedukti verification for the first time. This will enable systems in the Dedukti framework to invoke Leo-III to generate verified proofs in non-classical logics, and make verification of automated reasoning accessible to a number of non-classical logics for the first time. Future work that will yield the fully automatic output of verifiable Leo-III proofs in the Lambdapi format is outlined in the following.

7.2 Verification of Refutation

In the current implementation, the axioms as well as the negated conjecture are encoded as Lambdapi-axioms and, based on them, a term of type `Prf ⊥` is derived to show a contradiction. This representation follows the TSTP certificate and encodes the clauses resulting from individual inference rules as individual steps. However, proving the conjecture itself would also be possible in the Lambdapi encoding. In such an encoding, the proof would be composed into one long script following the pattern shown below:

```
1 symbol completeProof: Prf conjecture ≔
2 begin
3     have Contradiction : Prf(¬ conjecture) → Prf ⊥
4         {assume negatedConjecture;
5         // ...
6         };
7
8     refine npp conjecture (¬I (¬ conjecture) Contradiction)
9 end;
```

Rather than directly encoding the negated conjecture as a Lambdapi-axiom, this approach would show the contradiction based on the negated conjecture as an assumption. To this end, a subproof would have to be constructed to show that the negation of the conjecture to be proven allows us to derive a

proof of `⊥`. This would allow us to assume a term `negatedConjecture : Prf (¬ conjecture)` and then, in turn, provide the steps of the previous proof encoding as subproofs. `¬I` can then be used to map the term of type `Prf(¬ conjecture) → Prf ⊥` to a term of type `Prf(¬ ¬ conjecture)`. Proving the conjecture based on this is then a simple instantiation of `npp`. This way, not only the inference steps but also the approach of proving the conjecture by refutation would be verified. In the future development of the implementation, this proof encoding will thus be adopted.

7.3 Future work.

7.3.1 Correctness of the encoding

Proving the soundness of the encoding proposed in Section 3.2 is challenging and must be addressed in future work. A common approach involves proving both termination and confluence, and then arguing that these properties ensure the existence of a normal form that corresponds to a proof in natural deduction [6]. However, proving termination is difficult, though research on proof termination for super-consistent systems may present a promising option [32]. Existing proofs for related systems and alternative encoding approaches could also offer solutions. For instance, [35] suggests a new computational approach to encoding systems in Dedukti, resulting in encodings for which correctness is easier to prove by design. Another method for HOL is to encode it following the pure type system formalism. Minimal constructive HOL can be expressed as a pure type system, as proposed in [4], the encoding of which is shown to be sound in [28, 4]. The Holide project demonstrates an alternative encoding of HOL [5], differing from the theory U encoding by using equality as the primitive connective rather than implication and universal quantification.

7.3.2 Planned Extensions of the encoding.

The extended calculus EP of Leo-III comprises over 30 inference rules in total and the encoding presented here will have to be extended to this full set of rules. Apart from that, two restrictions were imposed on the encoding presented herein that will be lifted in future work. The theoretical encoding will therefore be extended in three steps:

I) Extension of the encoding to the full (extended) calculus EP.

II) Verification of the clausification steps, including skolemization.

III) The extension of the type system and inference rules with rank 1 polymorphism.

This will then be the theoretical foundation based on which the extensions of existing tools will be carried out to automatically verify HOL proofs in the Dedukti framework:

IV) The full implementation of automated proof encoding of Leo-III.

V) The extension of the tool GDV for HOL.

Fig. 18 sets the steps representing the theoretical work (**I** - **III**) necessary to automatize the encodings of Leo-III proofs into the context of the reasoning process of the Leo-III system and illustrates how the implementation of an automated encoding (**IV**) will result in verifiable proofs that can in turn be used to extend GDV-LP (**V**).

I. The specific requirements of the remaining inference rules have to be analyzed in a similar fashion as the ones presented herein and the developed verification and encoding strategies will have to be adapted accordingly. Furthermore, a representation of the choice operator present in Leo-III has to be added to the Lambdapi theory in order to encode the choice related rules. This will extend the work presented here to a complete foundational encoding of monomorphic HOL problems and proofs with an restriction to CNF.

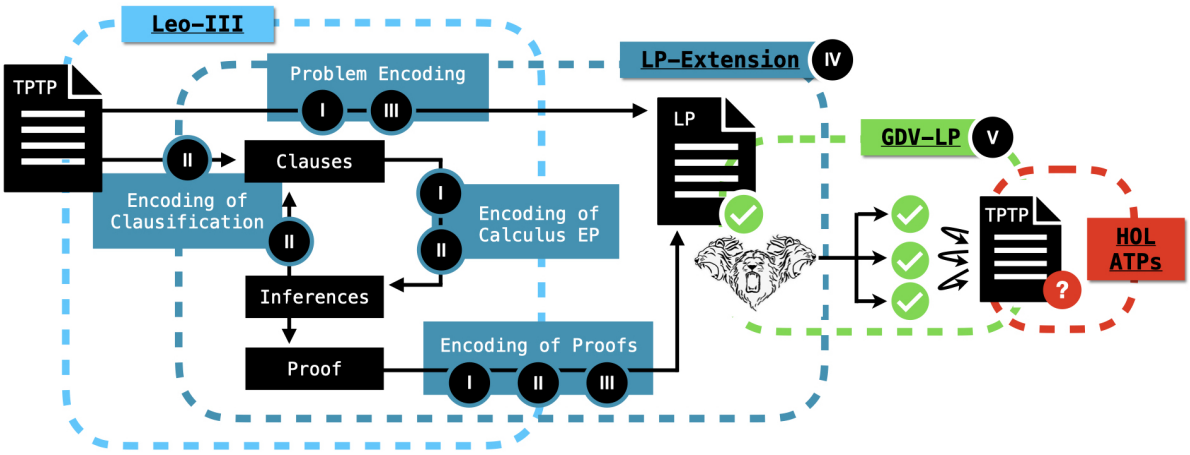

Figure 18: Next Steps in the Context of the Interacting Systems. The colored dotted lines divide the illustrated processes into the different systems: Leo-III is light blue, the Lambdapi extension is turquoise, GDP-LP is green and red signifies other ATP systems. The entities these systems operate on are given with a black backdrop and black arrows indicate steps of the reasoning process. The next steps are indicated by their respective numbers in black circles and placed on the arrow of their associated processes. Turquoise labels specify which steps are being encoded.

II. The verification approaches used in Dedukti aim at proving that derived formulas are logical consequences of their parents, which does not hold for skolemization, making the application of alternative strategies necessary. This problem is also encountered in FOL reasoning and has likewise come up in the implementation of some of the previously mentioned FOL tools [40, 66, 22]. *SKonverto*[9] is a tool designed to derive a Dedukti proof without Skolem terms from a FOL TPTP proof that involved skolemization. There is, however, a lack of currently available strategies for HOL in this respect, and the development, assessment, and implementation of such techniques will have merit beyond the scope of this project. The extension of SKonverto offers a promising prospect.

III. A further challenge is represented by the encoding of the polymorphism of Leo-III. Handling the specific restrictions it is governed by (type quantification only ranges over monomorphic types and can only be used at prenex position, type variables only range over monomorphic types and only monomorphic types are allowed as arguments to type applications [65]) will entail two steps: First, the current theory has to be extended with operators that quantify, apply and abstract over the monomorphic and polymorphic types. This has to be tailored to the specific restrictions to accurately represent the type system of Leo-III. Second, polymorphic types have to be unified to ensure their compatibility before rule applications. This additional step is governed by its own set of type unification rules, which will likewise have to be encoded, proven and included in the Lambdapi output.

IV. Together these objectives will provide the encodings and strategies necessary to produce verifiable Lambdapi proofs. The encoding will be automatized as an open source Lambdapi extension of Leo-III.

V. Upon the completion of the automated encoding, Leo-III can be used to verify proofs of other HOL systems outputting TSTP certificates. This can be achieved through the extension of GDV-LP [66] with the option to invoke Leo-III as a trusted prover.

[9] https://github.com/Deducteam/SKonverto

References

[1] P. B. Andrews. General models and extensionality. *The Journal of Symbolic Logic*, 37(2):395–397, 1972.

[2] P. B. Andrews. General models, descriptions, and choice in type theory. *J. Symb. Log.*, 37(2):385–394, 1972.

[3] P. B. Andrews. *An introduction to mathematical logic and type theory: to truth through proof*, volume 27. Springer Science & Business Media, 2013.

[4] A. Assaf. *A framework for defining computational higher-order logics*. PhD thesis, École polytechnique, 2015.

[5] A. Assaf and G. Burel. Translating HOL to dedukti. In C. Kaliszyk and A. Paskevich, editors, *Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015*, volume 186 of *EPTCS*, pages 74–88, 2015.

[6] 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 λΠ-calculus modulo theory. *CoRR*, abs/2311.07185, 2023.

[7] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge university press, 1998.

[8] H. Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[9] H. P. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.

[10] C. Benzmüller. *Equality and extensionality in automated higher order theorem proving*. PhD thesis, Saarland University, Saarbrücken, Germany, 1999.

[11] C. Benzmüller, C. E. Brown, and M. Kohlhase. Higher-order semantics and extensionality. *J. Symb. Log.*, 69(4):1027–1088, 2004.

[12] C. Benzmüller, C. E. Brown, and M. Kohlhase. Cut-simulation and impredicativity. *Log. Methods Comput. Sci.*, 5(1), 2009.

[13] C. Benzmüller and D. Miller. Automation of higher-order logic. In J. H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 215–254. Elsevier, 2014.

[14] C. Benzmüller and P. Andrews. Church's Type Theory. In E. N. Zalta and U. Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2024 edition, 2024.

[15] P. Blackburn, J. F. van Benthem, and F. Wolter. *Handbook of modal logic*. Elsevier, 2006.

[16] B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Found. Trends Priv. Secur.*, 1(1-2):1–135, 2016.

[17] F. Blanqui. Type safety of rewrite rules in dependent types. In Z. M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 13:1–13:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[18] F. Blanqui, G. Dowek, É. Grienenberger, G. Hondet, and F. Thiré. A modular construction of type theories. *Log. Methods Comput. Sci.*, 19(1), 2023.

[19] S. Bobzien. Ancient Logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2020 edition, 2020.

[20] M. Boespflug and G. Burel. Coqine: Translating the calculus of inductive constructions into the λΠ-calculus modulo. In D. Pichardie and T. Weber, editors, *Proceedings of the Second International Workshop on Proof Exchange for Theorem Proving, PxTP 2012, Manchester, UK, June 30, 2012*, volume 878 of *CEUR Workshop Proceedings*, pages 44–50. CEUR-WS.org, 2012.

[21] G. Burel. Experimenting with deduction modulo. In N. S. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 2011.

[22] G. Burel. A shallow embedding of resolution and superposition proofs into the λΠ-calculus modulo. In J. C. Blanchette and J. Urban, editors, *Third International Workshop on Proof Exchange for Theorem Proving, PxTP 2013, Lake Placid, NY, USA, June 9-10, 2013*, volume 14 of *EPiC Series in Computing*, pages 43–57. EasyChair, 2013.

[23] G. Bury. *Integrating rewriting, tableau and superposition into SMT. (Intégrer la réecriture, la méthode des tableaux et la superposition dans les solveurs SMT)*. PhD thesis, Sorbonne Paris Cité, France, 2019.

[24] G. Cantor. Über eine elementare frage der mannigfaltigkeitslehre, jahresbericht der dmv (vol. 1, pp. 75–78). references to cantor (1932). *English trans. in Ewald (1996)*, 2:278–280, 1892.

[25] A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[26] A. Coltellacci, S. Merz, and G. Dowek. Reconstruction of SMT proofs with lambdapi. In G. Reger and Y. Zohar, editors, *Proceedings of the 22nd International Workshop on Satisfiability Modulo Theories co-located with the 36th International Conference on Computer Aided Verification (CAV 2024), Montreal, Canada, July, 22-23, 2024*, volume 3725 of *CEUR Workshop Proceedings*, pages 13–23. CEUR-WS.org, 2024.

[27] T. Coquand and G. P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[28] D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2007.

[29] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[30] N. G. De Bruijn. The mathematical language automath, its usage, and some of its extensions. In *Studies in Logic and the Foundations of Mathematics*, volume 133, pages 73–100. Elsevier, 1994.

[31] D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, and O. Hermant. Zenon modulo: When achilles outruns the tortoise using deduction modulo. In K. L. McMillan, A. Middeldorp, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*, pages 274–290. Springer, 2013.

[32] G. Dowek. Models and termination of proof reduction in the lambda pi-calculus modulo theory. In I. Chatzigiannakis, P. Indyk, F. Kuhn, and A. Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 109:1–109:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[33] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *J. Autom. Reason.*, 31(1):33–72, 2003.

[34] G. Dowek and B. Werner. Proof normalization modulo. *J. Symb. Log.*, 68(4):1289–1316, 2003.

[35] T. Felicissimo. Adequate and computational encodings in the logical framework dedukti. In A. P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel*, volume 228 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[36] H. Freudenthal. Zur intuitionistischen deutung logischer formeln. *Compositio mathematica*, 4:112–116, 1937.

[37] H. Ganzinger and C. Benzmüller. Extensional higher-order paramodulation and rue-resolution. In *Automated Deduction—CADE-16: 16th International Conference on Automated Deduction Trento, Italy, July 7–10, 1999 Proceedings 16*, pages 399–413. Springer, 1999.

[38] G. Gentzen. Untersuchungen über das logische schließen. ii. *Mathematische zeitschrift*, 39, 1935.

[39] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38:173–198, 1931.

[40] M. Y. E. Haddad, G. Burel, and F. Blanqui. EKSTRAKTO A tool to reconstruct dedukti proofs from TSTP files (extended abstract). In G. Reis and H. Barbosa, editors, *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019*, volume 301 of *EPTCS*, pages 27–35, 2019.

[41] R. Harper, F. Honsell, and G. D. Plotkin. A framework for defining logics. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*, pages 194–204. IEEE Computer Society, 1987.

[42] L. Henkin. Completeness in the theory of types. *J. Symb. Log.*, 15(2):81–91, 1950.

[43] L. Horsten. Philosophy of Mathematics. In E. N. Zalta and U. Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2023 edition, 2023.

[44] W. A. Howard et al. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[45] G. P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.

[46] M. Janowicz, L. Ochnio, L. J. Chmielewski, and A. Orłowski. Application of automated theorem-proving to philosophical thought: Spinoza's ethics. In *Information Science and Applications 2017: ICISA 2017 8*, pages 512–518. Springer, 2017.

[47] J. W. Klop, V. Van Oostrom, and F. Van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical computer science*, 121(1-2):279–308, 1993.

[48] M. Kohlhase. *A mechanization of sorted higher-order logic based on the resolution principle*. PhD thesis, Saarland University, Saarbrücken, Germany, 1994.

[49] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.

[50] G. W. Leibniz and G. W. Leibniz. *Discourse on Metaphysics: 1686*. Springer, 1989.

[51] W. McCune. Solution of the robbins problem. *J. Autom. Reason.*, 19(3):263–276, 1997.

[52] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991.

[53] J. Moschovakis. Intuitionistic Logic. In E. N. Zalta and U. Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2023 edition, 2023.

[54] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. 2004.

[55] J. Otten. Mleancop: A connection prover for first-order modal logic. In S. Demri, D. Kapur, and C. Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, volume 8562 of *Lecture Notes in Computer Science*, pages 269–276. Springer, 2014.

[56] F. J. Pelletier and A. Hazen. Natural Deduction Systems in Logic. In E. N. Zalta and U. Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2024 edition, 2024.

[57] F. Portoraro. Automated Reasoning. In E. N. Zalta and U. Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2023 edition, 2023.

[58] M. Rathjen and W. Sieg. Proof Theory. In E. N. Zalta and U. Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2024 edition, 2024.

[59] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 298–313. Springer, 1983.

[60] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[61] S. Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2-3):111–126, 2002.

[62] A. Steen. *Extensional paramodulation for higher-order logic and its effective implementation Leo-III*. PhD thesis, Free University of Berlin, Dahlem, Germany, 2018.

[63] A. Steen and C. Benzmüller. Extensional higher-order paramodulation in leo-iii. *J. Autom. Reason.*, 65(6):775–807, 2021.

[64] A. Steen, G. Sutcliffe, T. Scholl, and C. Benzmüller. Solving modal logic problems by translation to higher-order logic. In A. Herzig, J. Luo, and P. Pardo, editors, *Logic and Argumentation - 5th International Conference, CLAR 2023, Hangzhou, China, September 10-12, 2023, Proceedings*, volume 14156 of *Lecture Notes in Computer Science*, pages 25–43. Springer, 2023.

[65] A. Steen, M. Wisniewski, and C. Benzmüller. Going polymorphic - TH1 reasoning for leo-iii. In T. Eiter, D. Sands, G. Sutcliffe, and A. Voronkov, editors, *IWIL@LPAR 2017 Workshop and LPAR-21 Short Presentations, Maun, Botswana, May 7-12, 2017*, volume 1 of *Kalpa Publications in Computing*, pages 100–112. EasyChair, 2017.

[66] G. Sutcliffe. Semantic derivation verification: Techniques and implementation. *Int. J. Artif. Intell. Tools*, 15(6):1053–1070, 2006.

[67] G. Sutcliffe. The TPTP World - Infrastructure for Automated Reasoning. In E. Clarke and A. Voronkov, editors, *Proceedings of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, number 6355 in Lecture Notes in Artificial Intelligence, pages 1–12. Springer-Verlag, 2010.

[68] G. Sutcliffe. The CADE ATP System Competition - CASC. *AI Magazine*, 37(2):99–101, 2016.

[69] G. Sutcliffe. The Logic Languages of the TPTP World. *Logic Journal of the IGPL*, 2022.

[70] M. Taprogge. Leo-iii-lp, 2024. https://zenodo.org/records/13889978.

[71] M. Taprogge and A. Steen. Flexible automation of quantified multi-modal logics with interactions. In *German Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 215–230. Springer, 2023.

[72] P. Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015.

[73] A. N. Whitehead and B. Russell. *Principia mathematica to* 56*, volume 2. Cambridge University Press, 1997.

A Construction Rules

```
// head symbol ∧

opaque symbol c∧r a b c : ( Prf b → Prf c) →  Prf(a ∧ b) → Prf(a ∧ c)  ≔
begin
    assume a b c h1 h2;
    have H1 : Prf a
        {refine ∧El a b h2};
    have H2 : Prf c
        {refine h1 (∧Er a b h2)};
    refine ∧I a c H1 H2;
end;

opaque symbol c∧l a b c : ( Prf a → Prf c) → Prf(a ∧ b) → Prf(c ∧ b)  ≔
begin
    assume a b c h1 h2;
    have H1 : Prf b
        {refine ∧Er a b h2};
    have H2 : Prf c
        {refine h1 (∧El a b h2)};
    refine ∧I c b H2 H1;
end;


// head symbol ∨

opaque symbol c∨r a b c : (Prf b → Prf c) → Prf(a ∨ b) → Prf(a ∨ c)  ≔
begin
    assume a b c h1 h2 b1 h3 h4;
    have H1: Prf b → Prf b1
        {assume h5;
        refine h4 (h1 h5)};
    refine ∨E a b b1 h3 H1 h2;
end;

opaque symbol c∨l a b c : (Prf a → Prf c) → Prf(a ∨ b) → Prf(c ∨ b)  ≔
begin
    assume a b c h1 h2 b1 h3 h4;
    have H1: Prf a → Prf b1
        {assume h5;
        refine h3 (h1 h5)};
    refine ∨E a b b1 H1 h4 h2;
end;


// head symbol ⇒

opaque symbol c⇒ a b c : (Prf b → Prf c) → Prf(a ⇒ b) → Prf(a ⇒ c) ≔
begin
    assume a b c h1 h2 h3;
    refine h1 (h2 h3)
end;


// head symbol ¬

```

```
opaque symbol c¬ x y : ((Prf y) → (Prf x)) → Prf(¬ x) → Prf (¬ y) ≔
begin
    assume x y h1 h2 h3 b;
    refine h2 (h1 h3) b;
end;


// head symbol =

opaque symbol eqTrainsitivity [T] (a b c : El T) : Prf(a = b) → Prf(b = c) → Prf
    (a = c) ≔
begin
    assume T a b c h1 h2 g h3T;
    have H1: Prf(g b)
        {refine h1 g h3T};
    refine h2 g H1
end;

opaque symbol c=l [T] a b c (f g : Prop → El T) : (Prf a → Prf c) → (Prf c → Prf
     a) → Prf((f a) = (g b)) → Prf((f c) = (g b))≔
begin
    assume T a b c f g h1 h2 h3;
    have H1 : Prf (c = a)
        {refine propExt c a h2 h1};
    have H2 : Prf ((f c) = (f a))
        {refine =def [o] c a H1 (λ z, (f z) = (f a)) (=ref [T] (f a))};
    refine eqTrainsitivity [T] (f c) (f a) (g b) H2 h3;
end;

opaque symbol c=r [T] a b c (f g : Prop → El T) : (Prf b → Prf c) → (Prf c → Prf
     b) → Prf((f a) = (g b)) → Prf((f a) = (g c))≔
begin
    assume T a b c f g h1 h2 h3;
    have H1 : Prf (b = c)
        {refine propExt b c h1 h2};
    have H2 : Prf ((g b) = (g c))
        {refine =def [o] b c H1 (λ z, (g z) = (g c)) (=ref [T] (g c))};
    refine eqTrainsitivity [T] (f a) (g b) (g c) h3 H2;
    proofterm;
end;


// Head symbol ∀

opaque symbol c∀ [T] (x : El T) a b : (Prf a → Prf b) → Prf(∀(λ x : El T, a) ) →
     Prf(∀(λ x : El T, b) ) ≔
begin
    assume T x a b h1 h2 h3;
    refine h1 (h2 x);
end;


// Head symbol ∃

opaque symbol c∃ [T] (x : El T) a b : (Prf a → Prf b) → Prf(∃(λ x : El T, a) ) →
     Prf(∃(λ x : El T, b) ) ≔
begin
```

```
108     simplify;
109     assume T x a b h1 h2 b1 h3;
110     have H1: Prf b
111         {refine h2 b (λ x , h1)};
112     refine h3 x H1
113 end;
114
115
116 // Head symbol λ
117
118 opaque symbol cλ q a b : (Prf a → Prf b) → (Prf b → Prf a) → Prf((λ x : El o, q
        x)a) → Prf((λ x : El o, q x)b) ≔
119 begin
120     assume q a b h1 h2 h3;
121     refine =def b a (propExt b a h2 h1) q h3;
122 end;
```

Note that the rules here are proven using the rewrite rules defining the connectives. The reason for this is that these rules are only used in the encodings that do not use the `rewrite` tactic and therefore the rewrite rules can be safely imported into the files containing the proof verification rather than being provided in an extra file like correctness.lp .

B Accessory Rules for Transforming to Equational Literals and back

```
opaque symbol posPropNegEq_eq x: (Prf (x = (¬ ((¬ x) = ⊤)))):=
begin
    assume x;
    refine propExt x (¬ ((¬ x) = ⊤)) _ _
    {assume h1;
    have H1: Prf((¬ x) = ⊤) → Prf ⊥
        {assume h2;
        refine ¬E x h1 ((=def [o] (¬ x) ⊤ h2 (λ z, z)) ⊤I)};
    refine ¬I ((¬ x) = ⊤) H1}
    {assume h1;
        refine ∨E x (¬ x) x _ _ (em x)
            {assume h2;
            refine h2}
            {assume h2;
            have H1: Prf((¬ x) = ⊤)
                {refine propExt (¬ x) ⊤ _ _
                    {assume h3;
                    refine ⊤I}
                    {assume h3;
                    refine h2}};
            refine ⊥E x (¬E ((¬ x) = ⊤) H1 h1)}}
end;

opaque symbol negPropPosEq_eq x: (Prf ((¬ x) = ((¬ x) = ⊤))):=
begin
    assume x;
    refine propExt (¬ x) ((¬ x) = ⊤) _ _
        {assume h1;
        refine propExt (¬ x) ⊤ _ _
            {assume h2;
            refine ⊤I}
            {assume h2;
            refine h1}}
        {assume h1;
        have H1: Prf((¬ x) = ⊤) → Prf(¬ x)
            {assume h2;
            refine (=def [o] (¬ x) ⊤ h2 (λ z, z)) ⊤I};
        refine H1 h1}
end;

opaque symbol negPropNegEq_eq x: (Prf ((¬ x) = (¬ (x = ⊤)))):=
begin
    assume x;
    refine propExt (¬ x) (¬ (x = ⊤)) _ _
        {assume h1;
        have H1: Prf(x = ⊤) → Prf ⊥
            {assume h2;
            refine ¬E x (=def [o] x ⊤ h2 (λ z, z) ⊤I) h1};
        refine ¬I  (x = ⊤) H1}
        {assume h1;
        refine ¬I x _;
        assume h2;
        have H1: Prf(x = ⊤)
```

```
            {refine propExt x ⊤ _ _
                {assume h3;
                refine ⊤I}
                {assume h3;
                refine h2}};
        refine ¬E (x = ⊤) H1 h1}
end;

opaque symbol topPosProp_eq x: (Prf (( ⊤ = x) = x)):=
begin
    assume x;
    refine propExt (⊤ = x) x _ _
        {assume h1;
        have H1 : Prf x → Prf x
            {assume h2;
            refine h2};
        refine (⇒E ⊤ x (=def ⊤ x h1 (λ z, z ⇒ x) (⇒I x x H1))) ⊤I}
        {assume h1;
        refine propExt ⊤ x _ _
            {assume h2;
            refine h1}
            {assume h2;
            refine ⊤I}}
end;

opaque symbol botNegProp_eq x: (Prf ((⊥ = x) = (¬ x))):=
begin
    assume x;
    refine propExt (⊥ = x) (¬ x) _ _
        {assume h1;
        have H1 : Prf x → Prf x
            {assume h2;
            refine h2};
        refine  ¬I x (⇒E x ⊥ (=def ⊥ x h1 (λ z, x ⇒ z) (⇒I x x H1)))}
        {assume h1;
        refine propExt ⊥ x _ _
            {assume h2;
            refine ⊥E x h2}
            {assume h2;
            refine ¬E x h2 h1}}
end;
```

C Simplification Rules

First, the proofs of simplification rules will be explained using (Simp7) as an example, then the remaining rules will be defined.

Generally, the proof of the equality literals will follow the same pattern: We will first assume the variables of the respective encoding and then use the Lambdapi-axiom `propExt`, which maps the proofs of a bidirectional implication of two terms to a proof of their equality. Both directions of the implication will then be shown in two subproofs. In the case of (Simp7), which allows us to omit $\bot$ from a disjunction ($(s \vee \bot) \to s$), the encoding can be proven as follows:

```
1  opaque symbol simp7_eq x: (Prf (x = (x ∨ ⊥))):=
2  begin
3      assume x;
4      refine propExt x (x ∨ ⊥) _ _
5          {assume h1;
6          refine ∨Il x ⊥ h1}
7          {assume h1;
8          refine ∨E x ⊥ x _ _ h1
9              {assume h2;
10             refine h2}
11             {assume h2;
12             refine ⊥E x h2}}
13 end;
```

The two subproofs we need to provide here are for `Prf x → Prf (x ∨ ⊥)` and the reverse direction, `Prf (x ∨ ⊥) → Prf x`. To prove the former, we assume `h1 : Prf x` (line 5), which we can use to prove the new focused goal `Prf (x ∨ ⊥)` using the introduction rule `∨Il` (line 6). In the second subproof, we assume `h1 : Prf (x ∨ ⊥)` (line 7) and use the elimination rule `∨E` and two subproofs demonstrating that `x` follows from both sides of the disjunction individually, to provide a term of type `Prf x` (line 8). For the left-hand side, we need to prove `Prf x → Prf x`, which can be achieved by simply assuming `h2 : Prf x` and refining with it (lines 9 and 10). For the right-hand side, the assumption of `h2 : Prf ⊥` can be mapped to an arbitrary proof using `⊥E`. An instantiation with `x` thus yields the desired proof (lines 11 and 12).

The proofs of the remaining simplification rules follow the same pattern:

```
1  opaque symbol simp1_eq x: (Prf (x = (x ∨ x))):=
2  begin
3      assume x;
4      refine propExt x (x ∨ x) _ _
5          {assume h2;
6          refine ∨Il x x h2}
7          {assume h1;
8          refine ∨E x x x _ _ h1
9              {assume h2;
10             refine h2}
11             {assume h2;
12             refine h2}}
13 end;
14
15 opaque symbol simp2_eq x: (Prf (x = (x ∧ x))):=
16 begin
17     assume x;
18     refine propExt x (x ∧ x) _ _
```

```
        {assume h1;
        refine ∧I x x h1 h1}
        {assume h1;
        refine ∧El x x h1}
end;

opaque symbol simp3_eq x: (Prf (⊤ = ((¬ x) ∨ x))):=
begin
    assume x;
    refine propExt ⊤ ((¬ x) ∨ x) _ _
        {assume h2;
        have em_sym: Prf(¬ x ∨ x)
        {refine ∨E x (¬ x) (¬ x ∨ x) _ _ (em x)
            {assume h3;
            refine ∨Ir (¬ x) x h3}
            {assume h3;
            refine ∨Il (¬ x) x h3}};
        refine em_sym}
        {assume h1;
        refine ⊤I}
end;

opaque symbol simp4_eq x: (Prf (⊥ = ((¬ x) ∧ x))):=
begin
    assume x;
    refine propExt ⊥ ((¬ x) ∧ x) _ _
        {assume h1;
        refine ⊥E (¬ x ∧ x) h1}
        {assume h1;
        refine ¬E x _ _
            {refine ∧Er (¬ x) x h1}
            {refine ∧El (¬ x) x h1}}
end;

opaque symbol simp5_eq x: (Prf (⊤ = (x ∨ ⊤))):=
begin
    assume x;
    refine propExt ⊤ (x ∨ ⊤) _ _
        {assume h2;
        refine ∨Ir x ⊤ h2}
            {assume h1;
        refine ⊤I}
end;

opaque symbol simp6_eq x: (Prf (x = (x ∧ ⊤))):=
begin
    assume x;
    refine propExt x (x ∧ ⊤) _ _
        {assume h1;
        refine ∧I x ⊤ h1 ⊤I}
        {assume h1;
        refine ∧El x ⊤ h1}
end;

opaque symbol simp8_eq x: (Prf (⊥ = (x ∧ ⊥))):=
begin
    assume x;
```

```
76      refine propExt ⊥ (x ∧ ⊥) _ _
77          {assume h1;
78          type ⊥E x h1;
79          refine ∧I x ⊥ (⊥E x h1) h1}
80          {assume h1;
81          refine ∧Er x ⊥ h1}
82  end;
83
84  opaque symbol simp9_eq T x: (Prf ( ⊤ = (x = x))):=
85  begin
86      assume T x;
87      refine propExt ⊤ (x = x) _ _
88          {assume h1;
89          refine =ref [T] x}
90          {assume h1;
91          refine ⊤I}
92  end;
93
94  opaque symbol simp10_eq T x: (Prf (⊥ = (¬ (x = x)))):=
95  begin
96      assume T x;
97      refine propExt ⊥ (¬ (x = x)) _ _
98          {assume h1;
99          refine ⊥E (¬ (x = x)) h1}
100         {assume h1;
101         have H1: Prf (x = x) → Prf ⊥
102             {assume h2;
103             refine ¬E (x = x) h2 h1};
104         refine H1 (=ref [T] x)}
105 end;
106
107 opaque symbol simp11_eq x: (Prf (x = (x = ⊤))):=
108 begin
109     assume x;
110     refine propExt x (x = ⊤) _ _
111         {assume h1;
112         refine propExt x ⊤ _ _
113             {assume h2;
114             refine ⊤I}
115             {assume h2;
116             refine h1}}
117         {assume h1;
118         refine (=def [o] x ⊤ h1 (λ z: Prop, z)) ⊤I}
119 end;
120
121 opaque symbol simp12_eq x: (Prf ((¬ x) = (¬(x = ⊤)))):=
122 begin
123     assume x;
124     refine propExt  (¬ x) (¬(x = ⊤)) _ _
125         {assume h1;
126         have H1: Prf(x = ⊤) → Prf ⊥
127             {assume h2;
128             have H1_1: Prf x
129                 {refine =def [o] x ⊤ h2 (λ z: Prop, z) ⊤I};
130             refine ¬E x H1_1 h1};
131         refine ¬I (x = ⊤) H1}
132         {assume h1;
```

```
        have H1: Prf x → Prf ⊥
            {assume h2;
            refine ¬E  (x = ⊤) _ h1;
            refine propExt x ⊤ _ _
                {assume h3;
                refine ⊤I}
                {assume h3;
                refine h2}};
        refine ¬I x H1}
end;

opaque symbol simp15_eq: (Prf (⊤ = (¬ ⊥))):=
begin
    refine propExt ⊤ (¬ ⊥) _ _
        {assume h1;
        have H1: Prf ⊥ → Prf ⊥
            {assume h2;
            refine h2};
        refine ¬I ⊥ H1}
        {assume h1;
        refine ⊤I}
end;

opaque symbol simp16_eq: (Prf (⊥ = (¬ ⊤))):=
begin
    refine propExt ⊥ (¬ ⊤) _ _
        {assume h1;
        refine ⊥E (¬ ⊤) h1}
        {assume h1;
        refine ¬E ⊤ ⊤I h1}
end;

opaque symbol simp17_eq x: (Prf (x = (¬ ¬ x))):=
begin
    assume x;
    refine propExt x (¬ ¬ x) _ _
        {assume h1;
        have H1: Prf(¬ x) → Prf ⊥
            {assume h2;
            refine ¬E x h1 h2};
        refine ¬I (¬ x) H1}
        {assume h1;
        refine npp x h1}
end;
```

D Calculus Rules

D.1 Functional Extensionality

```
1 opaque symbol PFE [T] [S] (f : (El (S ⇝ T))) (g : (El (S ⇝ T))) (x : (El S)): ((
      Prf (f = g)) → (Prf ((f x) = (g x)))) ≔
2 begin
3     assume T S f g x h;
4     refine =def [(S ⇝ T)] f g h (λ y, (y x) = (g x)) (=ref [T] (g x))
5 end;
```

To prove the implication of `(f x) = (g x)` by `f = g`, `h: Prf(f = g)` is assumed and used to prove `(f x) = (g x)` through the use of `=def`, the instantiation of which results in the type `Prf ((g x) = (g x)) → Prf ((f x) = (g x))`. To this term, we can apply the instantiation of `=ref`, which proves `(g x) = (g x)` (line 4).

D.2 Boolean Extensionality

Boolean Extensionality requires two proof terms representing the possible results for both (PBE), which is applied to positive equational literals, and (NBE), which is applied to negative ones.
(PBE). The first encoding is defined as:

```
1 opaque symbol PBE_r x y: Prf(x = y) → Prf(x ∨ (¬ y)) ≔
2 begin
3     assume x y h;
4     refine =def [o] x y h (λ z, z ∨ ¬ y) (em y);
5 end;
```

First, the variables `x` and `y` as well as the hypothesis `h1: Prf (x = y)` are assumed (line 3), leaving `Prf (x ∨ ¬ y)` as the focused goal. A term of that type is provided by first applying `x`, `y`, `h` and the anonymous function `(λ z, z ∨ ¬ y)` to `=def`, which results in a term of type `Prf (y ∨ ¬ y) → Prf (x ∨ ¬ y)`. The application of an instantiated version of the Lambdapi-axiom encoding the principle of the excluded middle, `em y` with type `Prf (y ∨ ¬ y)` therefore results in the required proof-term. The proof of the second rule is defined as follows.

```
1  opaque symbol PBE_l x y: Prf(x = y) → Prf((¬ x) ∨ y) ≔
2  begin
3      assume x y h;
4      have em_sym: Prf(¬ y ∨ y)
5          {refine ∨E y (¬ y) (¬ y ∨ y) _ _ (em y)
6              {assume h2;
7              refine ∨Ir (¬ y) y h2}
8              {assume h2;
9              refine ∨Il (¬ y) y h2}};
10     refine =def [o] x y h (λ z, ¬ z ∨ y) em_sym;
11 end;
```

The proof is analogous, the only difference is that we here require an additional step to show a permuted version of `em` (lines 4 to 9). This can be proven using the introduction and elimination rules of ∨ analogously to the proof provided in Sect. 4.3.3.

(NBE). The first version of the rule is defined as follows.

```
1  opaque symbol NBE_p x y: Prf(¬(x = y)) → Prf(x ∨ y):=
2  begin
3      assume x y h1;
4      refine ∨E x (¬ x) (x ∨ y) _ _ (em x)
5          {assume h2;
6          refine ∨Il x y h2}
7          {assume h2;
8          have H1: Prf y
9              {have H2: Prf (¬ y) → Prf ⊥
10                 {assume h3;
11                 have H3: Prf x → Prf y
12                     {assume h4;
13                     refine ⊥E y (¬E x h4 h2)};
14                 have H4: Prf y → Prf x
15                     {assume h4;
16                     refine ⊥E x (¬E y h4 h3)};
17                 refine ¬E (x = y) (propExt x y H3 H4) h1};
18             refine npp (y) (¬I (¬ y) H2)};
19         refine ∨Ir x y H1};
20 end;
```

After assuming `x`, `y` and `h1: Prf (¬ (x = y))`, the focused goal we are left with is `Prf (x ∨ y)`. We will follow a case distinction to construct a proof: We show in two subproofs that both `x` and `¬ x` imply `(x ∨ y)`. `∨E` can then be used to map these two proofs to `Prf (x ∨ ¬ x) → Prf (x ∨ y)`. Applying the instantiation of the Lambdapi-axiom `em` with `x`, resulting in a term of type `Prf (x ∨ ¬ x)`, then yields the desired proof-term. This is the `refine` step in line 4. The first subproof (lines 5-6) assumes `h2 : Prf x` and then shows `Prf (x ∨ y)`, once more using the introduction rule of ∨ (line 7). For the second subproof we assume `h2 : Prf (¬ x)` (line 8) and then need to show `Prf (x ∨ y)`. The idea we will follow to do so is to use the proof of the inequality of `x` and `y` provided by `h1` and the proof of `¬ x` provided by `h2` to construct a proof of `y`, that can in turn be used to prove our goal, `x ∨ y`, using the introduction rule for ∨ (line 19). To prove `y`, we first construct `H2 : Prf (¬ y → ⊥)` in a subproof (lines 9 -18) and then map it to a proof of `y` using the Lambdapi-axiom encoding double negation elimination, `npp` and the instantiation `¬I (¬ y) H2`, that results in type `¬ ¬ y` (line 18). This subproof of `H2 : Prf(¬ y → ⊥)` first assumes `h3 : Prf (¬ y)` and then needs to derive a contradiction in order to provide a term of type `Prf ⊥`. We will do this by constructing a term of type `x = y` based on the two proofs we have for $\neg x$ (`h2`) and $\neg y$ (`h3`), which we can then use together with `h1 : Prf (¬ (x = y))` to prove ⊥. To provide a proof of `x = y`, we once more first show the bidirectional implication, this is done in two subproofs of `H3: Prf x → Prf y` (lines 11 - 13) and `H4: Prf y → Prf x` (lines 14 - 16). Since these subproofs are analogous, we will only discuss the first one. Here, we first assume `h4 : Prf x` and then provide a proof of ⊥ by showing a contradiction arising from `h4` and `h2`, which proofs `¬ x`. This term is given by `(¬E x h4 h2)` and, since `⊥E` maps a proof of ⊥ an arbitrary proof-term, we can use it to prove our goal `Prf y`, this is done in line 14. Finally we can then instantiate `propExt` with `x`, `y`, `H3` and `H4` to obtain a term of type `Prf (x = y)` which, along with `h1` of type `Prf (¬ (x = y))`, we map to the desired `Prf ⊥` using `¬E` (line 17). As described above, the resulting proof of `¬ y → ⊥` is then used to prove `y` (line 18), which in turn is needed for the proof of `x ∨ y`.

The second encoding follows the same pattern:

```
1  opaque symbol NBE_n x y: Prf(¬(x = y)) → Prf(¬ x ∨ ¬ y):=
2  begin
3      assume x y h1;
4      refine ∨E x (¬ x) (¬ x ∨ ¬ y) _ _ (em x)
5          {assume h2;
6          have H1: Prf (y) → Prf ⊥
7              {assume h3;
8              have H2: Prf(x) → Prf(y)
9                  {assume h4;
10                 refine h3};
11             have H3: Prf(y) → Prf(x)
12                 {assume h4;
13                 refine h2};
14             refine ¬E (x = y) (propExt x y H2 H3) h1};
15         refine ∨Ir (¬ x) (¬ y) (¬I y H1)}
16         {assume h2;
17         refine ∨Il (¬ x) (¬ y) h2}
18 end;
```

Again, we provide a proof by showing that both `x` and `¬ x` imply `¬ x ∨ ¬ y` and simply proof the latter using of the introduction rule `∨Il` (lines 16 - 17). Here, we however have to prove `Prf(¬ x ∨ ¬ y)` based on the assumption `h1 : Prf (¬ (x = y))`, which is less complicated, as the proof of `¬ y` we need in place of the proof of `¬ ¬ y` of the previous case is easier to obtain and does not require the elimination of double negation. Furthermore, the proofs of `H2` and `H3` we require to prove `x = y` are more straightforward as we already have assumptions that provide proofs for `x` and `y` themselves rather than their negations (`h2` and `h3` respectively). Otherwise, the proof is analogous.

D.3 Equal Factoring

The encoding of (EqFact) for positive literals is...

```
1  opaque symbol EqFact_p [T] x y z v: ((Prf ((x = y) ∨ (z = v))) → (Prf ((x = y) ∨
       (¬ (x = z)) ∨ (¬ (y = v))))):=
2  begin
3      assume T x y z v h1;
4      refine (∨E (x = y) ( ¬ (x = y)) ((x = y) ∨ (¬ (x = z)) ∨ (¬ (y = v))) _ _ )
           (em (x = y))
5                  {assume h2;
6                  refine ∨Il (x = y) ((¬ (x = z)) ∨ (¬ (y = v))) h2}
7                  {assume h3;
8                  refine (∨E (x = z) ( ¬ (x = z)) ((x = y) ∨ (¬ (x = z)) ∨ (¬ (y =
                       v))) _ _ ) (em (x = z))
9                      {assume h4;
10                     refine (∨E (y = v) ( ¬ (y = v)) ((x = y) ∨ (¬ (x = z)) ∨ (¬
                           (y = v))) _ _ ) (em (y = v))
11                         {assume h5;
12                         have H1: Prf (z = v)
13                             {refine ∨E (x = y) (z = v) (z = v) _ _ h1
14                                 {assume h6;
15                                 refine ⊥E (z = v) (¬E (x = y) h6 h3)}
16                                 {assume h7;
17                                 refine h7}};
18                         have H2: Prf(x = v)
19                             {refine =def [T] x z h4 (λ a, (a = v)) H1};
20                         have H3: Prf(v = y)
```

```
21                                  {refine =def [T] y v h5 (λ a, (v = a)) (=ref [T] v)
                                        };
22                              have H4: Prf(x = y)
23                                  {refine =def [T] x v H2 (λ a, (a = y)) H3};
24                              refine ⊥E ((x = y) ∨ (¬ (x = z)) ∨ (¬ (y = v))) (¬E  (x
                                    = y) H4 h3)}
25                              {assume h8;
26                              refine ∨Ir (x = y) ((¬ (x = z)) ∨ (¬ (y = v))) (∨Ir (¬ (
                                    x = z)) (¬ (y = v)) h8)}}
27                          {assume h9;
28                          refine ∨Ir (x = y) ((¬ (x = z)) ∨ (¬ (y = v))) (∨Il (¬ (x =
                                z)) (¬ (y = v)) h9)}}
29  end;
```

After assuming `T`, `x`, `y`, `z`, `v`, and `h1 : Prf ((x = y) ∨ (z = v))`, we are left with the goal of proving `Prf ((x = y) ∨ (¬ (x = z)) ∨ (¬ (y = v)))`. To construct this proof, we will perform a series of case distinctions based on the equality relations between `x`, `y`, `z`, and `v`:

We start by instantiating `∨E` as shown in line 4, which results in a term mapping terms of types `Prf (x = y) → Prf (x = y)`, `Prf (¬ (x = y)) → Prf (x = y)` and `Prf ((x = y) ∨ (¬ (x = y)))` to the desired proof-term. The instantiated Lambdapi-axiom of excluded middle (`em (x = y)`) provides the last argument, the first two implications are shown in subproofs. These represent the first case differentiation between `(x = y)` and `¬ (x = y)`. The first subproof is straight forward, as it simply assumes `h2 : Prf (x = y)` and uses it to prove the goal `Prf ((x = y) ∨ (¬ (x = z)) ∨ (¬ (y = v)))` with `∨Il` (line 6).

For the other subproof, we assume `h3 : Prf (¬ (x = y))` and proceed to the next case distinction, this time for `x = z` through the analogous application of `∨E` and `em` (line 8). This again results in two subproofs representing the two cases and requiring us to prove `(x = y) ∨ (¬ (x = z)) ∨ (¬ (y = v))` based on the assumptions of `h4 : Prf (x = z)` (line 9) and `h9 : Prf (¬ (x = z))` (line 27) respectively. The latter is again shown by a nested application of `∨Ir` and `∨Il` to first construct a proof-term of `¬ (x = z) ∨ ¬ (y = v)` based on `h9` and then, based on that, proof `(x = y) ∨ (¬ (x = z)) ∨ (¬ (y = v))` (line 28).

For the other subproof, we conduct the last case differentiation, this time for `y = v` (line 10). The subproof for `Prf (¬ (y = v)) → Prf (x = y ∨ (¬ (x = z) ∨ ¬ (y = v)))` is again constructed by assuming `h8 : Prf (¬ (y = v))` (line 25) and applying nested instances of the introduction rules `∨Il` and `∨Ir` (line 26). The last subproof now uses the assumptions made in the case distinctions to prove `x = y`, which forms a contradiction with `h3 : Prf (¬ (x = y))` and can hence be used to derive `Prf ⊥`, that in turn can be used to draw an arbitrary conclusion and hence can be applied to the instantiated elimination rule `⊥E` to serve as a proof-term for `(x = y) ∨ (¬ (x = z)) ∨ (¬ (y = v))` (line 24). Proving `x = y` involves several steps: We first construct a term `H1` of type `Prf (z = v)` by showing that it follows from both literals of `h1 : Prf ((x = y) ∨ (z = v))` and refining with `∨E` (line 13). The subproofs are provided by deriving a contradiction of the assumed `h6 : Prf (x = y)` and `h3 : Prf (¬ (x = y))` for the first literal (lines 14 - 15). The second literal corresponds to the assumption of the second subproof (`h7 : Prf (z = v)`) and is therefore provided as a proof (lines 16 - 17). Furthermore, we proof `H3: Prf(v = y)` based on `h5: Prf(y = v)` as discussed in Sec. 4.3.2. We now have the following three terms at our disposal: `h4 : Prf (x = z)`, `H1 : Prf (z = v)` and `H3: Prf(v = y)`. Proving `x = y` thus reduces to the task of proving transitivity of `=` in two instances (lines 18 - 19 and lines 22 - 23). In both cases, this is done with an instantiating of `=def`. This completes the last step of the proof.

Since the proof for the negative version of the rule can be proven analogously, it is not explicitly given here.

E Leo-III TSTP Output for Cantor Subjectivity

```
thf(sk1_type,type,
    sk1: $i > $i > $o ).

thf(sk2_type,type,
    sk2: ( $i > $o ) > $i ).

thf(1,conjecture,
    ~ ? [A: $i > $i > $o] :
      ! [B: $i > $o] :
      ? [C: $i] :
        ( ( A @ C )
        = B ),
    file('sur_cantor.p',sur_cantor) ).

thf(2,negated_conjecture,
    ~ ~ ? [A: $i > $i > $o] :
        ! [B: $i > $o] :
        ? [C: $i] :
          ( ( A @ C )
          = B ),
    inference(neg_conjecture,[status(cth)],[1]) ).

thf(3,plain,
    ~ ~ ? [A: $i > $i > $o] :
        ! [B: $i > $o] :
        ? [C: $i] :
          ( ( A @ C )
          = B ),
    inference(defexp_and_simp_and_etaexpand,[status(thm)],[2]) ).

thf(4,plain,
    ? [A: $i > $i > $o] :
    ! [B: $i > $o] :
    ? [C: $i] :
      ( ( A @ C )
      = B ),
    inference(polarity_switch,[status(thm)],[3]) ).

thf(5,plain,
    ! [A: $i > $o] :
      ( ( sk1 @ ( sk2 @ A ) )
      = A ),
    inference(cnf,[status(esa)],[4]) ).

thf(6,plain,
    ! [A: $i > $o] :
      ( ( sk1 @ ( sk2 @ A ) )
      = A ),
    inference(lifteq,[status(thm)],[5]) ).

thf(7,plain,
```

```
    ! [B: $i,A: $i > $o] :
      ( ( sk1 @ ( sk2 @ A ) @ B )
      = ( A @ B ) ),
    inference(func_ext,[status(esa)],[6]) ).

thf(9,plain,
    ! [B: $i,A: $i > $o] :
      ( ( sk1 @ ( sk2 @ A ) @ B )
      | ~ ( A @ B ) ),
    inference(bool_ext,[status(thm)],[7]) ).

thf(250,plain,
    ! [B: $i,A: $i > $o] :
      ( ( sk1 @ ( sk2 @ A ) @ B )
      | ( ( A @ B )
       != ( ~ ( sk1 @ ( sk2 @ A ) @ B ) ) )
      | ~ $true ),
    inference(eqfactor_ordered,[status(thm)],[9]) ).

thf(270,plain,
    ( sk1
    @ ( sk2
      @ ^ [A: $i] :
          ~ ( sk1 @ A @ A ) )
    @ ( sk2
      @ ^ [A: $i] :
          ~ ( sk1 @ A @ A ) ) ),
    inference(pre_uni,[status(thm)],[250:
    [bind(A,$thf( ^ [C: $i] : ~ ( sk1 @ C @ C ) )),
    bind(B,$thf( sk2 @ ^ [C: $i] : ~ ( sk1 @ C @ C ) ))]]) ).

thf(8,plain,
    ! [B: $i,A: $i > $o] :
      ( ~ ( sk1 @ ( sk2 @ A ) @ B )
      | ( A @ B ) ),
    inference(bool_ext,[status(thm)],[7]) ).

thf(18,plain,
    ! [B: $i,A: $i > $o] :
      ( ~ ( sk1 @ ( sk2 @ A ) @ B )
      | ( ( A @ B )
       != ( ~ ( sk1 @ ( sk2 @ A ) @ B ) ) )
      | ~ $true ),
    inference(eqfactor_ordered,[status(thm)],[8]) ).

thf(32,plain,
    ~ ( sk1
      @ ( sk2
        @ ^ [A: $i] :
            ~ ( sk1 @ A @ A ) )
      @ ( sk2
        @ ^ [A: $i] :
```

```
            ~ ( sk1 @ A @ A ) ) ),
    inference(pre_uni,[status(thm)],[18:
    [bind(A,$thf( ^ [C: $i] : ~ ( sk1 @ C @ C ) )),
    bind(B,$thf( sk2 @ ^ [C: $i] : ~ ( sk1 @ C @ C ) ))]]) ).

thf(372,plain,
    $false,
    inference(rewrite,[status(thm)],[270,32]) ).

thf(373,plain,
    $false,
    inference(simp,[status(thm)],[372]) ).
```

F Leo-III Lambdapi Output for Cantor Subjectivity

```
require open thesis_lpFiles.extt thesis_lpFiles.naturalDeduction thesis_lpFiles.accessoryRules thesis_lpFiles.calcRules thesis_lpFiles.simp;
// OBJECT DECLARATIONS ////////////////////////////////////

symbol sk1: (El (ι ⇝ (ι ⇝ o)));
symbol sk2: (El ((ι ⇝ o) ⇝ ι));


// PROBLEM ENCODING ///////////////////////////////////////

symbol negatedConjecture0: ((Prf ((¬ (¬ (∃(λ (A : (El (ι ⇝ (ι ⇝ o)))), (∀(λ (B : (El (ι ⇝ o))), (∃(λ (C : (El ι)), ((A C) = B))))))))))));


// PROOF ENCODING /////////////////////////////////////////

// PolaritySwitch
opaque symbol step4: ((Prf ((∃(λ (A : (El (ι ⇝ (ι ⇝ o)))), (∀(λ (B : (El (ι ⇝ o))), (∃(λ (C : (El ι)), ((A C) = B)))))))))) ≔
begin
  have PolaritySwitch0 : (Prf ((∃(λ (A : (El (ι ⇝ (ι ⇝ o)))), (∀(λ (B : (El (ι ⇝ o))), (∃(λ (C : (El ι)), ((A C) = B))))))) = (¬ (¬ (∃(λ (A : (El (ι ⇝ (ι ⇝ o)))), (∀(λ (B : (El (ι ⇝ o))), (∃(λ (C : (El ι)), ((A C) = B)))))))))))
    {refine (simp17_eq (∃(λ (A : (El (ι ⇝ (ι ⇝ o)))), (∀(λ (B : (El (ι ⇝ o))), (∃(λ (C : (El ι)), ((A C) = B))))))))};
  rewrite PolaritySwitch0;
  refine (negatedConjecture0)
end;

// The rule leo.modules.calculus.RenameCNF$@783a467b is not encoded yet
symbol step5 : (Π (A : (El (ι ⇝ o))), (Prf (((sk1 (sk2 A)) = A))));

// FuncExt
opaque symbol step7: (Π (B : (El ι)), Π (A : (El (ι ⇝ o))), (Prf (((sk1 (sk2 A) B) = (A B))))) ≔
begin
  assume B A;
  have PFE_0 : ((Prf ((sk1 (sk2 A)) = A)) → (Prf ((sk1 (sk2 A) B) = (A B))))
    {refine (PFE (sk1 (sk2 A)) A B)};
  have FunExtApplication : (Prf (((sk1 (sk2 A) B) = (A B))))
    {refine (PFE_0 (step5 A))};
  refine (FunExtApplication)
end;

// BoolExt
opaque symbol step9: (Π (B : (El ι)), Π (A : (El (ι ⇝ o))), (Prf ((sk1 (sk2 A) B) ∨ (¬ (A B))))) ≔
begin
  assume B A;
  refine ((PBE_r (sk1 (sk2 A) B) (A B)) (step7 B A))
end;

// OrderedEqFac
opaque symbol step252: (Π (B : (El ι)), Π (A : (El (ι ⇝ o))), (Prf ((sk1 (sk2 A) B) ∨ (¬ ((A B) = (¬ (sk1 (sk2 A) B)))) ∨ (¬ ⊤)))) ≔
```

```
47 begin
48   assume B A;
49   have WholeEqFactStep : ((Prf ((sk1 (sk2 A) B) ∨ (¬ (A B)))) → (Prf ((sk1 (sk2
       A) B) ∨ (¬ ((A B) = (¬ (sk1 (sk2 A) B)))) ∨ (¬ ⊤))))
50     {assume h1;
51     have TransformToEqLits : (Prf ((¬ ((¬ (sk1 (sk2 A) B)) = ⊤)) ∨ (¬ ((A B) = ⊤
         ))))
52       {rewrite .[x in (x ∨ _)] (negEqPosProp_eq (sk1 (sk2 A) B));
53       rewrite .[x in (_ ∨ x)] (negEqNegProp_eq (A B));
54       refine ((step9 B A))};
55     have EqFact : (Prf ((¬ ((¬ (sk1 (sk2 A) B)) = ⊤)) ∨ (¬ ((¬ (sk1 (sk2 A) B))
         = (A B))) ∨ (¬ (⊤ = ⊤))))
56       {refine ((EqFact_n [o] (¬ (sk1 (sk2 A) B)) ⊤ (A B) ⊤) TransformToEqLits)};
57     have EqSymmetry : (Prf ((¬ ((¬ (sk1 (sk2 A) B)) = ⊤)) ∨ (¬ ((A B) = (¬ (sk1
         (sk2 A) B)))) ∨ (¬ (⊤ = ⊤))))
58       {rewrite .[x in (_ ∨ ¬ x ∨ _)] (eqSym_eq [o]);
59       refine (EqFact)};
60     have TransformToNonEqLits : (Prf ((sk1 (sk2 A) B) ∨ (¬ ((A B) = (¬ (sk1 (sk2
          A) B)))) ∨ (¬ ⊤)))
61       {rewrite .[x in (x ∨ _ ∨ _)] posPropNegEq_eq;
62       rewrite .[x in (_ ∨ _ ∨ x)] negPropNegEq_eq;
63       refine (EqSymmetry)};
64     refine (TransformToNonEqLits)};
65   refine (WholeEqFactStep (step9 B A))
66 end;
67
68 // PreUni
69 opaque symbol step265: ((Prf ((sk1 (sk2 (λ (A : (El ι)), (¬ (sk1 A A)))) (sk2 (λ
       (A : (El ι)), (¬ (sk1 A A)))))))) ≔
70 begin
71   have Substitution : (Prf ((sk1 (sk2 (λ (C : (El ι)), (¬ (sk1 C C)))) (sk2 (λ (
       C : (El ι)), (¬ (sk1 C C))))) ∨ (¬ (((λ (C : (El ι)), (¬ (sk1 C C))) (sk2
       (λ (C : (El ι)), (¬ (sk1 C C))))) = (¬ (sk1 (sk2 (λ (C : (El ι)), (¬ (sk1
       C C)))) (sk2 (λ (C : (El ι)), (¬ (sk1 C C)))))))) ∨ (¬ ⊤)))
72     {refine (step252 (sk2 (λ (C : (El ι)), (¬ (sk1 C C)))) (λ (C : (El ι)), (¬ (
         sk1 C C))))};
73   rewrite simp7_eq;
74   rewrite .[x in (_ ∨ x)] (simp10_eq o ((λ (C : (El ι)), (¬ (sk1 C C))) (sk2 (λ
       (C : (El ι)), (¬ (sk1 C C))))));
75   rewrite .[x in (_ ∨ x)] simp7_eq;
76   rewrite .[x in (_ ∨ _ ∨ x)] simp16_eq;
77   refine (Substitution)
78 end;
79
80 // BoolExt
81 opaque symbol step8: (Π (B : (El ι)), Π (A : (El (ι ⇝ o))), (Prf ((¬ (sk1 (sk2 A
     ) B)) ∨ (A B)))) ≔
82 begin
83   assume B A;
84   refine ((PBE_1 (sk1 (sk2 A) B) (A B)) (step7 B A))
85 end;
86
87 // OrderedEqFac
88 opaque symbol step18: (Π (B : (El ι)), Π (A : (El (ι ⇝ o))), (Prf ((¬ (sk1 (sk2
     A) B)) ∨ (¬ ((A B) = (¬ (sk1 (sk2 A) B)))) ∨ (¬ ⊤)))) ≔
89 begin
90   assume B A;
```

```
  have WholeEqFactStep : ((Prf ((¬ (sk1 (sk2 A) B)) ∨ (A B))) → (Prf ((¬ (sk1 (
      sk2 A) B)) ∨ (¬ ((A B) = (¬ (sk1 (sk2 A) B)))) ∨ (¬ ⊤))))
    {assume h1;
    have TransformToEqLits : (Prf (((¬ (sk1 (sk2 A) B)) = ⊤) ∨ ((A B) = ⊤)))
      {rewrite .[x in (x ∨ _)] posEqNegProp_eq;
      rewrite .[x in (_ ∨ x)] (posEqPosProp_eq (A B));
      refine ((step8 B A))};
    have EqFact : (Prf (((¬ (sk1 (sk2 A) B)) = ⊤) ∨ (¬ ((¬ (sk1 (sk2 A) B)) = (A
        B))) ∨ (¬ (⊤ = ⊤))))
      {refine ((EqFact_p [o] (¬ (sk1 (sk2 A) B)) ⊤ (A B) ⊤) TransformToEqLits)};
    have EqSymmetry : (Prf (((¬ (sk1 (sk2 A) B)) = ⊤) ∨ (¬ ((A B) = (¬ (sk1 (sk2
        A) B)))) ∨ (¬ (⊤ = ⊤))))
      {rewrite .[x in (_ ∨ ¬ x ∨ _)] (eqSym_eq [o]);
      refine (EqFact)};
    have TransformToNonEqLits : (Prf ((¬ (sk1 (sk2 A) B)) ∨ (¬ ((A B) = (¬ (sk1
       (sk2 A) B)))) ∨ (¬ ⊤)))
      {rewrite .[x in (x ∨ _ ∨ _)] negPropPosEq_eq;
      rewrite .[x in (_ ∨ _ ∨ x)] negPropNegEq_eq;
      refine (EqSymmetry)};
    refine (TransformToNonEqLits)};
  refine (WholeEqFactStep (step8 B A))
end;

// PreUni
opaque symbol step33: ((Prf ((¬ (sk1 (sk2 (λ (A : (El ι)), (¬ (sk1 A A)))) (sk2
    (λ (A : (El ι)), (¬ (sk1 A A)))))))))) ≔
begin
  have Substitution : (Prf ((¬ (sk1 (sk2 (λ (C : (El ι)), (¬ (sk1 C C)))) (sk2 (
      λ (C : (El ι)), (¬ (sk1 C C)))))) ∨ (¬ (((λ (C : (El ι)), (¬ (sk1 C C))) (
      sk2 (λ (C : (El ι)), (¬ (sk1 C C))))) = (¬ (sk1 (sk2 (λ (C : (El ι)), (¬ (
      sk1 C C)))) (sk2 (λ (C : (El ι)), (¬ (sk1 C C)))))))) ∨ (¬ ⊤)))
    {refine (step18 (sk2 (λ (C : (El ι)), (¬ (sk1 C C)))) (λ (C : (El ι)), (¬ (
       sk1 C C))))};
  rewrite simp7_eq;
  rewrite .[x in (_ ∨ x)] (simp10_eq o ((λ (C : (El ι)), (¬ (sk1 C C))) (sk2 (λ
      (C : (El ι)), (¬ (sk1 C C))))));
  rewrite .[x in (_ ∨ x)] simp7_eq;
  rewrite .[x in (_ ∨ _ ∨ x)] simp16_eq;
  refine (Substitution)
end;

// RewriteSimp
opaque symbol step373: ((Prf (⊥))) ≔
begin
  have TransformToEqLits : (Prf (⊥ = (sk1 (sk2 (λ (A : (El ι)), (¬ (sk1 A A))))
      (sk2 (λ (A : (El ι)), (¬ (sk1 A A)))))))
    {rewrite botNegProp_eq;
    refine (step33)};
  rewrite TransformToEqLits;
  refine (step265)
end;
```