A Shallow Embedding of Resolution and Superposition Proofs into the $\lambda\Pi$-Calculus Modulo*

Guillaume Burel
ÉNSIE/Cédric
1 square de la résistance, 91025 Évry cedex, France

Abstract

The $\lambda\Pi$-calculus modulo is a proof language that has been proposed as a proof standard for (re-)checking and interoperability. Resolution and superposition are proof-search methods that are used in state-of-the-art first-order automated theorem provers. We provide a shallow embedding of resolution and superposition proofs in the $\lambda\Pi$-calculus modulo, thus offering a way to check these proofs in a trusted setting, and to combine them with other proofs. We implement this embedding as a backend of the prover iProver Modulo.

Introduction

Proof assistants have now achieved a quite high degree of maturity, and are able to certify rather big projects. One can for instance cite the certified compiler CompCert by Coq [14], or the seL4 microkernel specification in Isabelle/HOL [12]. Nevertheless, some of the current challenges concerning proof assistants are to overcome their lack of automation, and to help them cooperate better to share proof developments. A way of making proof assistants more automated is to delegate proof obligation to external automated theorem provers. This is for instance what the Sledgehammer [3] subsystem of Isabelle/HOL does, which passes on proof obligations to first-order automated theorem prover such as E or SPASS, or SMT solvers like CVC3, Yices or Z3.

To keep confidence in the whole proof, the question arises of the combination of the proof found by the automated prover and the rest of the proof-assistant development. For Sledgehammer, this is done by reproving the proof obligation with an Isabelle/HOL tactics, namely Metis, only keeping the information of which lemmas were needed by the automated prover to find the proof and searching the proof again from scratch using only these lemmas. Of course, it would be more interesting to directly retrieve the proof of the automated prover and to translate it into an Isabelle/HOL proof. However, automated theorem do not often output proofs, and when they do, it is not trivial to translate them into a proof assistant format. Furthermore, such a translation would have to be performed for each pair automated prover/proof assistant.

Another solution would be to have a single, universal proof format in which every part of a big proof would be translated and combined. An analogy can be drawn with the interoperability of programming languages, that are translated into an assembly language in which the linking is performed. Ideally, this universal standard for proofs should have the following properties: It should be simple, so that it should be easy to write a proof checker in which one could therefore have a high degree of confidence. Moreover, it should be expressive enough to be able to embed the basic logics of all theorem provers and proof assistants available. To help proof recombination, this embeddings should also be shallow. Although there is to the author’s knowledge no precise definition of what a shallow embedding is, it can be distinguished from

*This work is supported by the French ANR project BWare.
A Shallow Embedding of Resolution and Superposition Proofs into the \(\lambda\Pi\)-Calculus Modulo

Coq
OpenTheory
Focalize
iProver Modulo

Dedukti
HOL Light
HOL4

Figure 1: Dedukti as a universal proof language

a deep embedding by the fact that it reuses the features of the target language. For instance, connectives are translated as connectives, and not as constants, and the same for variables, binders, computations, etc. Now suppose that we have two input proof languages \(A\) and \(B\), with respective embeddings \(|·|_A\) and \(|·|_B\) into our target standard. We would like to combine a proof of \(P \Rightarrow Q\) in \(A\) and a proof of \(P\) in \(B\) to get a proof of \(Q\). Using deep embeddings, it would be hard to relate the translation \(|P \Rightarrow Q|_A\) and \(|P|_B\), that could a priori have nothing in common. On the contrary, using a shallow embedding, \(|P \Rightarrow Q|_A\) would be equal to \(|P|_A \Rightarrow |Q|_A\), where \(\Rightarrow\) is the implication of the target language. Therefore, it only remains to relate \(|P|_A\) and \(|P|_B\) which should be easier.

The \(\lambda\Pi\)-calculus modulo \([1, 5]\) is a proposed standard for proof interoperability. It is relatively simple, and an already efficient interpreter for it takes only a few hundred lines of code. The \(\lambda\Pi\)-calculus modulo is an extension of the \(\lambda\Pi\)-calculus, a proof language for minimal first-order logic also known as LF, \(\lambda P\), etc \([11]\). In the \(\lambda\Pi\)-calculus modulo, it is possible to have shallow embeddings of higher-order logics, what is not possible in pure \(\lambda\Pi\)-calculus. Cousineau and Dowek \([7]\) have shown that any pure type systems can be shallowly embedded into the \(\lambda\Pi\)-calculus modulo, including for instance the Calculus of Construction which serves as basis of the proof assistant Coq. Assaf \([1]\) has proved that simple type theory, the higher-order logic that is the foundation of proof assistants of the HOL family, can also be translated in the \(\lambda\Pi\)-calculus modulo in a shallow way. The \(\lambda\Pi\)-calculus modulo is therefore naturally a good candidate for a universal standard for proofs. Following this idea, a language called Dedukti\(^1\) was designed to declare proofs of the \(\lambda\Pi\)-calculus modulo, and a proof checker for this language, namely \texttt{dkparse}, was implemented. \texttt{dkparse} is available at \url{https://www.rocq.inria.fr/deducteam/Dedukti/}.

Tools related to Dedukti also include a translator of Coq proofs to Dedukti, namely CoqInE \([4]\) \url{http://www.ensiie.fr/~guillaume.burel/blackandwhite_coqInE.html.en}, and a translator from OpenTheory proofs (a standard for proofs of the HOL family) to Dedukti, namely Holide \([1]\) \url{https://www.rocq.inria.fr/deducteam/Holide/}. There exists also a prototype of a backend of the certifying programming environment FoCaLiZe to Dedukti, namely FoCalide \url{https://www.rocq.inria.fr/deducteam/Focalide/}. Figure \([1]\) summarizes the current tools available around Dedukti.

Current state-of-the-art automated theorem provers for first-order logic are based on the superposition calculus \([2]\), which can be seen as an extension of the resolution method \([10]\). This includes for instance the provers Vampire \([15]\), SPASS \([10]\) or E \([17]\). To be able to combine proofs from these provers with the developments of a proof assistant, we therefore want to translate them in the \(\lambda\Pi\)-calculus in a shallow manner. In this paper, we show how

\(^1\)“Dedukti” means “to deduce” in Esperanto.
A Shallow Embedding of Resolution and Superposition Proofs into the λΠ-Calculus Modulo

Empty

\[ \emptyset \text{ WF} \]

Declaration

\[ \Gamma \text{ WF} \quad \dfrac{\Gamma \vdash A : s \quad x \not\in \Gamma}{\Gamma, x : A \text{ WF}} \quad s \in \{\text{Type, Kind}\} \]

Sort

\[ \Gamma \text{ WF} \quad \dfrac{\Gamma \vdash \text{Type} : \text{Kind}}{} \]

Variable

\[ \Gamma \text{ WF} \quad \dfrac{x : A \in \Gamma}{\Gamma \vdash x : A} \]

Product

\[ \Gamma \vdash A : \text{Type} \quad \dfrac{\Gamma \vdash \Pi x : A. B : s}{\Gamma \vdash \Pi x : A. B : s} \quad s \in \{\text{Type, Kind}\} \]

Application

\[ \Gamma \vdash \Pi x : A. B \quad \dfrac{\Gamma \vdash U : A}{\Gamma \vdash (T U) : \{U/x\}B} \]

Abstraction

\[ \Gamma \vdash A : \text{Type} \quad \dfrac{\Gamma, x : A \vdash B : s \quad \Gamma, x : A \vdash T : B}{\Gamma \vdash \lambda x : A. T : \Pi x : A. B} \quad s \in \{\text{Type, Kind}\} \]

Conversion

\[ \Gamma \vdash T : A \quad \dfrac{\Gamma \vdash A : s \quad \Gamma \vdash B : s}{\Gamma \vdash (T U) : \{U/x\}B} \quad s \in \{\text{Type, Kind}\} \quad \text{and} \quad \Gamma \vdash A \equiv_{\beta} B \]

Figure 2: Type System for the λΠ-calculus

this is possible. We also present an implementation of this translation in the prover iProver Modulo, which is therefore able to produce proofs in Dedukti’s format.

In next section, we present formally the λΠ-calculus modulo. In Section 2, we describe the shallow embedding of first-order logic with equality in the λΠ-calculus modulo. Section 3 details the translation of resolution and superposition proofs. Its implementation in iProver Modulo is outlined in Section 4.

1 The λΠ-Calculus Modulo

The λΠ-calculus modulo [7, 5] is an extension of the λΠ-calculus, a proof language for minimal first-order logic also known as LF, λP, etc [11]. The λΠ-calculus is based on the Curry-Howard-DeBruijn correspondence, which means that proofs are represented by λ-terms and formulas by their types, and it can be seen as one of the simplest coherent Pure Type System, which means that there is no syntactic distinction between terms and types.

Pre-terms in the λΠ-calculus are defined by the grammar

\[ M, N, A, B ::= x \mid \lambda x : A. M \mid \Pi x : A. B \mid M \cdot N \mid \text{Type} \mid \text{Kind} \]

where \( x \) is an element of an infinite set of variables. A context is a set of couples of variables and pre-terms. A pre-term will be called a term when it is well-typed in the type system of Figure 2, where the judgment “\( \Gamma \text{ WF} \)” means that a context \( \Gamma \) is well-formed, and the judgment \( \Gamma \vdash T : A \) must be read as “\( T \) has type \( A \) in the context \( \Gamma \)”.

In the Conversion rule of the λΠ-calculus, \( A \equiv_{\beta} B \) means that \( A \) and \( B \) are \( \beta \)-convertible.

In the λΠ-calculus modulo, this conversion rule is extended by means of well-typed rewriting rules:

Definition 1 (Rewrite rule). A rewrite rule is a quadruple \( \Delta : l \mapsto A r \) composed of a context \( \Delta \) and three terms \( l, r \) and \( A \). It is well typed in a context \( \Gamma \) if:

- the context \( \Gamma, \Delta \) is well-formed;
• \( \Gamma, \Delta \vdash l : A \) and \( \Gamma, \Delta \vdash r : A \) are derivable judgments.

Intuitively, \( \Delta \) contains the type of the free variables of \( l \) and \( r \), and \( A \) ensures that \( l \) and \( r \) have the same type, what warrants the preservation of types through rewriting. In the rewrite rules that we use in the following, \( \Delta \) and \( A \) can often be inferred from \( l \) and \( r \), in which case we will omit them and simply write \( l \mapsto r \). As usual, a term \( t \) is rewritten by a rewrite rule \( l \mapsto r \) to a term \( s \) if their exists a substitution \( \theta \) such that a subterm of \( t \) at a position \( p \) is equal to \( \theta(l) \) and \( s \) is equal to \( t \) where the subterm at position \( p \) is replaced by \( \theta(r) \).

In the \( \lambda \Pi \)-calculus modulo, contexts can contain rewriting rules, and the type system of the \( \lambda \Pi \)-calculus is therefore extended by a new rule adding a well-typed rewriting rule in a context:

\[
\text{Rewrite} \quad \Gamma, \Delta \vdash l : A \quad \Gamma, \Delta \vdash r : A \\
\Gamma', (\Delta : l \mapsto^A r) \text{ WF}
\]

Given a context \( \Gamma \), we let \( \equiv_\Gamma \) be the smallest congruence generated by \( \beta \)-reduction and the rewriting rules of \( \Gamma \). The conversion rule of the \( \lambda \Pi \)-calculus is then replaced by the following one:

\[
\text{Conversion} \quad \Gamma \vdash T : A \quad \Gamma \vdash A : s \quad \Gamma \vdash B : s \quad s \in \{\text{Type, Kind}\} \text{ and } A \equiv_\Gamma B
\]

The case of the \( \lambda \Pi \)-calculus without modulo is regained when the contexts do not contain any rewriting rules.

A file in Dedukti’s format is a declaration of a context of the \( \lambda \Pi \)-calculus modulo. Syntactically, \( \lambda x : a . t \) and \( \Pi x : a . b \) are respectively written \( x : a \Rightarrow t \) and \( x : a \rightarrow b \), and a rewriting rule \( \Delta : l \mapsto r \) is declared as \( [\Delta] \ 1 \rightarrow^r \ 2 \). The tool \texttt{dkparse} checks that such a context is well-formed, in particular it checks that rewriting rules are well-typed. If in the context there is a declaration of a constant \( c \) of type \( A \), and a rule rewriting \( c \) into a term \( t \), the fact that the context is well-formed implies that \( t \) has type \( A \), and by the Curry-Howard correspondence this means that \( t \) is a proof of \( A \). Similarly, if a constant of type \( B \) is declared, but it is not rewritten, this can be seen as assuming the axiom \( B \).

## 2 Translating First-Order Logic in \( \lambda \Pi \)-Calculus Modulo

### 2.1 Deep and Shallow Embedding of First-Order Logic

This section is based on Dorra’s work [8], which itself borrows ideas from the embedding of pure type systems in the \( \Pi \)-calculus modulo [7].

We use standard definitions for terms, predicates, first-order propositions (with connectives \( \bot, \lor, \land, \lor \) and quantifiers \( \forall, \exists \)) as can be found in [10].

The translation of first-order logic in the \( \lambda \Pi \)-calculus modulo consists of two embeddings, one deep \( | \cdot | \) and one shallow \( || \cdot || \), that are linked by a decoding function \texttt{proof} that is defined by means of rewriting rules.

To define the deep embedding, we first define two constants \( \iota \) and \( \omicron \) of type \( \text{Type} \) that contains respectively the translation of terms and propositions. We add constants \( \bot : o, \top : o \rightarrow o, \land : o \rightarrow o \rightarrow o, \lor : o \rightarrow o \rightarrow o, \forall : (t \rightarrow o) \rightarrow o, \exists : (t \rightarrow o) \rightarrow o \) for the translation of connectives and quantifiers. For each function symbol \( f \) of arity \( n \) we add a constant \( f : \underbrace{\iota \rightarrow \cdots \rightarrow \iota}_n \rightarrow \iota \), and for each predicate symbol \( p \) of arity \( n \) we add a constant \( p : \underbrace{\iota \rightarrow \cdots \rightarrow \iota}_n \rightarrow o \). In a context \( X_1 : \iota, \ldots, X_m : \iota \) where \( X_1, \ldots, X_m \) are the free variables of
a formula $A$, we can then translate formulas by induction:

$$|X| = X$$

if $X$ is a variable.

$$|f(t_1, \ldots, t_n)| = f |t_1| \cdots |t_n|$$

$$|p(t_1, \ldots, t_n)| = \hat{p} |t_1| \cdots |t_n|$$

$$|\bot| = \bot$$

$$|\neg A| = \prime |A|$$

$$|A \Rightarrow B| = \Rightarrow |A| |B|$$

$$|A \lor B| = \lor |A| |B|$$

$$|A \land B| = \land |A| |B|$$

$$|\forall X.A| = \forall (\lambda X : t. |A|)$$

$$|\exists X.A| = \exists (\lambda X : t. |A|)$$

The shallow embedding is defined by $||A|| = \text{proof} |A|$ where $\text{proof}$ is a decoding function of type $o \rightarrow \text{Type}$. What makes this translation shallow is the definition of the decoding function by means of rewriting rules, that relates the deep embedding of connectives with their counterparts in $\lambda\Pi$-calculus modulo. $\Rightarrow$ is for instance related with $\rightarrow$, $\forall$ with $\Pi$, whereas the other connectives are related with their impredicative encoding in $\Pi$. We can add a constant $p$ of type $\tau \rightarrow \cdots \rightarrow \tau \rightarrow \text{Type}$ to get a shallow embedding of each predicate symbol $p$ whose arity is $n$. The rules defining $\text{proof}$ are therefore:

$$\text{proof } (\hat{p} t_1 \cdots t_n) \iff p t_1 \cdots t_n$$

$$\text{proof } \bot \rightarrow P : a. \text{proof } b$$

$$\text{proof } (\prime A) \rightarrow P : a. \text{proof } A \rightarrow \text{proof } b$$

$$\text{proof } (\Rightarrow A B) \rightarrow \text{proof } A \rightarrow \text{proof } B$$

$$\text{proof } (\lor A B) \rightarrow P : a. (\text{proof } A \rightarrow \text{proof } b) \rightarrow (\text{proof } B \rightarrow \text{proof } b) \rightarrow \text{proof } b$$

$$\text{proof } (\land A B) \rightarrow P : a. (\text{proof } A \rightarrow \text{proof } B \rightarrow \text{proof } b) \rightarrow \text{proof } b$$

$$\text{proof } (\forall f) \rightarrow P X : t. \text{proof } (f X)$$

$$\text{proof } (\exists f) \rightarrow P : a. (P X : t. \text{proof } (f X) \rightarrow \text{proof } b) \rightarrow \text{proof } b$$

where $b$ is a variable that does not appear in any first-order formula to avoid capture.

It can be proved that this translation is sound, that is that if a formula $A$ is provable in intuitionistic first-order logic, then there exists a term of type $||A||$ in the $\lambda\Pi$-calculus modulo the environment described above. It is also a conservative extension of intuitionistic first-order logic, in the sense that for all first-order formula $A$, if the type $||A||$ is inhabited in the environment defined above, then $A$ is provable in intuitionistic first-order logic.

Resolution and superposition are proof-search methods for first-order logic. They manipulate clauses. A literal is either an atomic formula (i.e. a predicate symbol applied to as many terms as its arity) or the negation of an atomic formula. A clause is a list of literals $L_1; \cdots ; L_n$. It corresponds to the formula $\forall X_1. \cdots \forall X_n. L_1 \lor \cdots \lor L_m$, where $X_1, \ldots, X_n$ are the free variables of $L_1, \ldots, L_n$. To ease the translation of resolution and superposition proofs, we translate clauses directly into a shallow embedding: A clause $L_1; \cdots ; L_m$ is translated as

$$||L_1; \cdots ; L_m|| = \Pi X_1 : t. \ldots \Pi X_n : t. P : a. [L_1]_b \rightarrow \cdots \rightarrow [L_m]_b \rightarrow \text{proof } b$$
where \(X_1, \ldots, X_n\) are the free variables in the clause and \([P]_\emptyset = [\lvert P \rvert \to \text{proof } b]\) for a positive literal \(P\) and \([-P]_\emptyset = ([\lvert P \rvert \to \text{proof } b]) \to \text{proof } b\) for a negative literal \(-P\). The empty clause is therefore translated as \(\Pi b : o. \text{proof } b\), which is also the translation of \(\bot\) as expected. It can be shown that the translation of a clause \(L_1; \cdots; L_m\) is implied by the translation of the corresponding formula \(\forall X_1. \ldots \forall X_n. L_1 \lor \cdots \lor L_m\). To get the other direction, one needs a classical axiom, for instance in the case of a clause containing only one literal.

### 2.2 Equality

The equality predicate \(\simeq\) is so pervasive that it is often useful to have a specific treatment of it. For instance, the resolution methods was extended into the superposition method to handle the equality better. To have a shallower translation of first-order logic with equality in the \(\lambda\Pi\)-calculus modulo, it is possible to define the equality predicate using Leibniz law.

\[
\simeq : t \to t \to \text{Type} \quad \simeq \to \lambda x : t. \lambda y : t. \Pi p : (t \to o). \text{proof } (p x) \to \text{proof } (p y)
\]

Usual properties of equality can then be proved, which avoid us to add them as axioms. For instance, reflexivity is proved by:

\[
\text{refl} : \Pi x : t. \simeq x x
\]

Commutativity has the following proof:

\[
\text{comm} : \Pi x : t. \Pi y : t. \simeq x y \to \simeq y x
\]

#### 3 Translating resolution and superposition proofs

### 3.1 Resolution

A derivation in resolution \cite{16} tries to refute a set of clauses by inferring new clauses by means of the two following inference rules, until the empty clause is derived.

\[
\text{Resolution} \quad \frac{P; C \quad -Q; D}{\sigma(C; D)} \quad \text{Factoring} \quad \frac{L; K; C}{\sigma(L; C)} \quad \sigma = \text{mgu}(P, Q)
\]

To translate resolution proofs, we decompose these rules into two steps: one instantiation step and one propositional step:

\[
\text{Instantiation} \quad \frac{C}{\sigma(C)} \quad \text{Identical Resolution} \quad \frac{P; C}{C; D} \quad \text{Identical Factoring} \quad \frac{L; L; C}{L; C}
\]

Of course these rules are applied modulo commutativity of \(;\), which means that \(P\) or \(L\) is not necessarily the first literal of the clauses.

Given some input clauses \(C_1, \ldots, C_k\), an identical-resolution derivation is a sequence of clauses \(C_1, \ldots, C_k, C_{k+1}, \ldots, C_n\) such that each clauses \(C_i\) for \(i > k\) is inferred from clauses among \(C_1, \ldots, C_{i-1}\) using one of the three rules above. The input set of clauses is shown unsatisfiable if \(C_n\) is the empty clause. To translate such a derivation in the \(\lambda\Pi\)-calculus modulo, we declare a constant \(c_i\) of type \(\lvert C_i \rvert\) for each \(1 \leq i \leq n\), and we add some rewriting rules to define
the constants $c_j$ for $k < j \leq n$. These rewriting rules depend on the rule used to infer $C_j$, and they use the constants corresponding to the clauses from which $C_j$ is inferred. At the end, since all other constants are defined, the only axioms are $||C_i||$ for $1 \leq i \leq k$, and the translation of the empty clause, that is $\forall \beta. \text{proof } \beta$ is proved from these axioms. This shows that the set of input clauses is indeed refuted.

To understand the translation of the inference rules, one needs to look at the computational content of terms that have as type the translation of a clause $L_1; \cdots; L_m$; intuitively, they are functions that take as arguments $n$ first-order terms to instantiate the free variables of the clause, a proposition $\beta$ to be proved, $m$ functions that given a term of type $||L_i||$ return a proof of $\beta$, and that return a proof of $\beta$.

The translation of the instantiation rule is relatively easy, since one just need to apply the image of the variable to the original clause, and to abstract over the new free variables:

\[
\text{Instantiation } \quad \frac{L_1; \cdots; L_m}{\sigma(L_1); \cdots; \sigma(L_m)}
\]

\[
c : \Pi x_1 : t. \cdots ; \Pi x_n : t. \Pi b : o. [L_1]_b \rightarrow \cdots \rightarrow [L_m]_b \rightarrow \text{proof } \beta
\]

\[
d : \Pi y_1 : t. \cdots ; \Pi y_k : t. \Pi b : o. [\sigma(L_1)]_b \rightarrow \cdots \rightarrow [\sigma(L_m)]_b \rightarrow \text{proof } \beta
\]

\[
d \mapsto \lambda y_1 : t. \cdots \lambda y_k : t. \sigma(x_1) \cdots (\sigma(x_n))
\]

The translation of factoring is also rather simple, since we just need to merge two literals:

\[
\text{Identical Factoring } \quad \frac{L_1; \cdots; L_i; \cdots; L_m}{L_1; \cdots; L_i; \cdots; L_m}
\]

\[
c : \Pi x_1 : t. \cdots ; \Pi x_n : t. \Pi b : o. [L_1]_b \rightarrow \cdots \rightarrow [L_i]_b \rightarrow \cdots \rightarrow [L_m]_b \rightarrow \text{proof } \beta
\]

\[
d : \Pi x_1 : t. \cdots ; \Pi x_n : t. \Pi b : o. [L_1]_b \rightarrow \cdots \rightarrow [L_i]_b \rightarrow \cdots \rightarrow [L_m]_b \rightarrow \text{proof } \beta
\]

\[
d \mapsto \lambda x_1 : t. \cdots \lambda x_n : t. \lambda b : o. \lambda l_1 : [L_1]_b. \cdots \lambda l_m : [L_m]_b. c \; x_1 \cdots x_n \; b \; l_1 \cdots l_i \cdots l_m
\]

To translate a resolution step, we can use the atom $P$ and its negation to get the proof of $\beta$. More precisely, we can use as term of type $[P]_b = [\|P\|] \rightarrow \text{proof } \beta$ in the translation of the clause $L_1; \cdots; P; \cdots; L_m$ the function that take a term $tp$ of type $[\|P\|]$ and that returns the clause $M_1; \cdots; \neg P; \cdots; M_m$ where the term for type $[\neg P]_b = ([\|P\|] \rightarrow \text{proof } \beta) \rightarrow \text{proof } \beta$ is the function that take a term $tnp$ of type $[\|L_1\|] \rightarrow \text{proof } \beta$ and return $tnp \; tp$, which is of type $\beta$.

\[
\text{Identical Resolution } \quad \frac{L_1; \cdots; L_i-1; P; L_i; \cdots; L_m \quad M_1; \cdots; M_{i-1}; \neg P; M_i; \cdots; M_l}{L_1; \cdots; L_m; M_1; \cdots; M_l}
\]

\[
c_1 : \Pi x_1 : t. \cdots ; \Pi x_n : t. \Pi b : o. [L_1]_b \rightarrow \cdots \rightarrow [P]_b \rightarrow \cdots \rightarrow [L_m]_b \rightarrow \text{proof } \beta
\]

\[
c_2 : \Pi y_1 : t. \cdots ; \Pi y_k : t. \Pi b : o. [M_1]_b \rightarrow \cdots \rightarrow [\neg P]_b \rightarrow \cdots \rightarrow [M_l]_b \rightarrow \text{proof } \beta
\]

\[
d : \Pi z_1 : t. \cdots ; \Pi z_j : t. \Pi b : o. [L_1]_b \rightarrow \cdots \rightarrow [L_m]_b \rightarrow [M_1]_b \rightarrow \cdots \rightarrow [M_l]_b \rightarrow \text{proof } \beta
\]

\[
d \mapsto \lambda z_1 : t. \cdots \lambda z_j : t. \lambda b : o. \lambda l_1 : [L_1]_b. \cdots \lambda l_m : [L_m]_b.
\]

\[
\lambda m_1 : [M_1]_b. \cdots \lambda m_l : [M_l]_b.
\]

\[
c_1 \; x_1 \cdots x_n \; b \; l_1 \cdots l_{i-1}
\]

\[
(\lambda tp : [\|P\|] \rightarrow \text{proof } \beta). \; c_2 \; y_1 \cdots y_k \; b \; m_1 \cdots m_{i-1}
\]

\[
(\lambda ntp : ([\|P\|] \rightarrow \text{proof } \beta). \; tnp \; tp) \; m_i \cdots m_l
\]

\[
l_i \cdots l_m
\]
Example 1. We want to refute the set of two clauses $p(X,Y); p(X,a)$ and $\neg p(b,Y)$. A possible derivation of the empty clause in resolution is the following:

1. $p(X,Y); p(X,a)$
2. $\neg p(b,Y)$
3. $p(X,a)$ applying Factor on 1
4. $\top \top$ applying Resolution on 2 and 3

If we decompose the instantiation from the inferences, we get

1. $p(X,Y); p(X,a)$
2. $\neg p(b,Y)$
3. $p(X,a); p(X,a)$ applying Instantation on 1 with $\sigma = \{Y \mapsto a\}$
4. $p(X,a)$ applying Identical Factoring on 3
5. $p(b,a)$ applying Instantation on 4 with $\sigma = \{X \mapsto b\}$
6. $\neg p(b,a)$ applying Instantation on 2 with $\sigma = \{Y \mapsto a\}$
7. $\top \top$ applying Identical Resolution on 5 and 6

We have a binary predicate symbol $p$ and two constants $a$ and $b$. The context of the translation in the $\lambda\Pi$-calculus modulo is therefore

\[
\begin{align*}
\iota & : \text{Type} \\
o & : \text{Type} \\
\text{proof} & : o \to \text{Type} \\
p & : \iota \to \iota \to o \\
p & : \iota \to \iota \to \text{Type} \\
\text{proof} (\hat{p} x y) & \mapsto p x y \\
a & : \iota \\
b & : \iota
\end{align*}
\]

We first declare the two input clauses:

\[
\begin{align*}
c_1 & : \Pi X : \iota. \Pi Y : \iota. \Pi b : o. (p X Y \rightarrow \text{proof } b) \rightarrow (p X a \rightarrow \text{proof } b) \rightarrow \text{proof } b \\
c_2 & : \Pi Y : \iota. \Pi b : o. ((p b Y \rightarrow \text{proof } b) \rightarrow \text{proof } b) \rightarrow \text{proof } b
\end{align*}
\]

We then declare the inferred clauses and define them as explained above:

\[
\begin{align*}
c_3 & : \Pi X : \iota. \Pi b : o. (p X a \rightarrow \text{proof } b) \rightarrow (p X a \rightarrow \text{proof } b) \rightarrow \text{proof } b \\
c_3 & \mapsto \lambda X : \iota. c_1 X a \\
c_4 & : \Pi X : \iota. \Pi b : o. (p X a \rightarrow \text{proof } b) \rightarrow \text{proof } b \\
c_4 & \mapsto \lambda X : \iota. \lambda b : o. \lambda l : (p X a \rightarrow \text{proof } b). c_3 X \mapsto l l \\
c_5 & : \Pi b : o. (p b a \rightarrow \text{proof } b) \rightarrow \text{proof } b \\
c_5 & \mapsto c_4 b \\
c_6 & : \Pi b : o. ((p b a \rightarrow \text{proof } b) \rightarrow \text{proof } b) \rightarrow \text{proof } b \\
c_6 & \mapsto c_2 a \\
c_7 & : \Pi b : o. \text{proof } b \\
c_7 & \mapsto \lambda b : o. c_5 b (\lambda tp : p b a. c_6 b (\lambda tnp : p b a \rightarrow \text{proof } b. tnp tp))
\end{align*}
\]
3.2 Superposition

Superposition can be seen as an extension of resolution to handle equality better. Superposition primarily uses four inference rules \((u \neq v \text{ denotes } \neg(u \equiv v))\):

**Equality Resolution** \[
\frac{u \neq v; R}{\sigma(R)} \quad \sigma = \text{mgu}(u,v)
\]

**Negative Superposition** \[
\frac{s \equiv t; S \quad u \neq v; R}{\sigma(u[t]_p \neq v; S; R)} \quad \sigma = \text{mgu}(u,p,s)
\]

**Positive Superposition** \[
\frac{s \equiv t; S \quad u \equiv v; R}{\sigma(u[t]_p \equiv v; S; R)} \quad \sigma = \text{mgu}(u,p,s)
\]

**Equality Factoring** \[
\frac{s \equiv t; u \equiv v; R}{\sigma(t \neq v; u \equiv v; R)} \quad \sigma = \text{mgu}(s,u)
\]

These rules are given with many conditions that restrict the cases when they can be applied. That makes the superposition calculus usable in practice in contrast to former paramodulation-based methods. Since we are only concerned in translating a proof, not finding one, these restrictions do not concern us.

Also, superposition-based provers use simplification rules, in which a set of clauses is replaced by another set of clauses. This too is not problematic for us since these simplification rules can in most of the cases be decomposed into the application of the four basic inference rules followed by the elimination of redundant clauses. Notable exceptions are the rules introducing and applying definitions in for instance the prover E, that we will not consider here.

Here again, to ease the translation, we will consider an explicit instantiation step and propositional rules:

**Identical Equality Resolution** \[
\frac{u \neq w; R}{u \neq w; R}
\]

**Negative Replacement** \[
\frac{s \equiv t; S \quad u[s]_p \neq v; R}{u[t]_p \neq v; S; R}
\]

**Positive Replacement** \[
\frac{s \equiv t; S \quad u[s]_p \equiv v; R}{u[t]_p \equiv v; S; R}
\]

**Identical Equality Factoring** \[
\frac{s \equiv t; s \equiv v; R}{t \neq v; s \equiv v; R}
\]

Once more, these rules can be applied modulo commutativity of ; and ≈. For ≈, it can be taken into account using the \text{comm} term (see Section 2.2). For simplicity, we assume in the following that equalities are oriented appropriately.

Since reflexivity is provable thanks to our encoding of equality, **Identical Equality Resolution** is rather easy to translate. Indeed, a term of type \([u \neq u]_s = (\vdash u \equiv u \Rightarrow \text{proof } b) \Rightarrow \text{proof } b\) can be \(\lambda p : (\|(u \equiv u)| \Rightarrow \text{proof } b|). p (\text{refl } u)\).

**Identical Equality Resolution** \[
\frac{L_1; \ldots; L_{i-1}; u \neq w; L_i \ldots; L_m}{L_1; \ldots; L_m}
\]

\[
c : \Pi x_1 : t. \ldots. \Pi x_n : t. \Pi b : o. [[L_1]][b] \Rightarrow \cdots \Rightarrow [[u \neq w]][b] \Rightarrow \cdots \Rightarrow [[L_m]][b] \Rightarrow \text{proof } b
\]

\[
d : \Pi y_1 : t. \ldots. \Pi y_k : t. \Pi b : o. [[L_1]][b] \Rightarrow \cdots \Rightarrow [[L_m]][b] \Rightarrow \text{proof } b
\]

\[
d \Rightarrow \lambda y_1 : t. \ldots. \lambda y_k : t. \lambda b : o. \lambda \lambda : [L_1][b] \Rightarrow \cdots \Rightarrow [L_m][b]
\]

\[
c x_1 \ldots x_n \, b \, \overline{l_1 \ldots l_{i-1} (\lambda p : (\|(u \equiv u)| \Rightarrow \text{proof } b|). p \, (\text{refl } u)) \, l_i \ldots l_m}
\]

For **Identical Equality Factoring**, we somehow need to refute \(s \equiv t\) from \(s \equiv v\) and \(t \neq v\). If we consider a term \(p\) of type \([t \neq v]_s = (\vdash t \equiv v \Rightarrow \text{proof } b) \Rightarrow \text{proof } b\), a term \(q\) of type \([s \equiv v]_s = \vdash s \equiv v\Rightarrow \text{proof } b\) and a term \(r\) of type \([s \equiv s]_s = \vdash s \equiv t\), the term \(p \, (\lambda z : t. \Rightarrow (\vdash z \equiv v \, b) \, q)\) has type \(b\).

**Identical Equality Factoring** \[
\frac{L_1; \ldots; L_{i-1}; s \equiv t; L_i; \ldots; L_m}{t \neq v; s \equiv v; L_1; \ldots; L_m}
\]

\[
c : \Pi x_1 : t. \ldots. \Pi x_n : t. \Pi b : o. [[L_1]][b] \Rightarrow \cdots \Rightarrow [[s \equiv t]][b] \Rightarrow \cdots \Rightarrow [[s \equiv v]][b] \Rightarrow \cdots \Rightarrow [[L_m]][b] \Rightarrow \text{proof } b
\]

\[
d : \Pi y_1 : t. \ldots. \Pi y_k : t. \Pi b : o. [[t \neq v]][b] \Rightarrow [[s \equiv v]][b] \Rightarrow [[L_1]][b] \Rightarrow \cdots \Rightarrow [[L_m]][b] \Rightarrow \text{proof } b
\]
A Shallow Embedding of Resolution and Superposition Proofs into the Mi-Calculus Modulo Burel

Example 2. We want to refute the set of three clauses $c \simeq g(a) \equiv X \simeq f(b, Y)$ and $g(Z) \simeq f(X, Z)$ and $g(c) \not\simeq g(f(X, Y))$. A possible derivation of the empty clause in superposition (without considering ordering restrictions) is the following:

For Positive Replacement, we can use the following idea: given a term $p$ of type $\|u[t]_p \simeq v\| \rightarrow \text{proof } \#a$, a term $q$ of type $\|u[s]_p \simeq v\|$ and a term $r$ of type $\|s \simeq t\|$, the term $p \ (r (\lambda z. \simeq \ |u[z]_p| \ |v|) \ q)$ has type $\text{proof } \#b$.

Positive Replacement

\[
\begin{array}{c}
L_1; \cdots; L_{i-1}; s \simeq t; L_i; \cdots; L_m \rightarrow M_1; \cdots; M_{h-1}; u[s]_p \simeq v; M_h; \cdots; M_t \\
\end{array}
\]

\[
\begin{array}{c}
c_1 : \Pi x_1 : t. \cdots \Pi x_n : t. \Pi b : o. \ \Pi L_1 : b \rightarrow \cdots \rightarrow \|s \simeq t\| \rightarrow \cdots \rightarrow \|L_m\| \rightarrow \text{proof } \#b \\
c_2 : \Pi y_1 : t. \cdots \Pi y_n : t. \Pi b : o. \ \Pi M_1 : b \rightarrow \cdots \rightarrow \|u[s]_p \simeq v\| \rightarrow \cdots \rightarrow \|M_t\| \rightarrow \text{proof } \#b \\
d : \Pi z_1 : t. \cdots \Pi z_j : t. \Pi b : o. \ \Pi u[t]_p \simeq v \rightarrow \|L_1\| \rightarrow \cdots \rightarrow \|L_m\| \rightarrow \|M_1\| \rightarrow \cdots \rightarrow \|M_t\| \rightarrow \text{proof } \#b \\
d \rightarrow \lambda z_1 : t. \cdots \lambda z_j : t. \lambda b : o. \ \lambda p : \|u[t]_p \simeq v\|. \ \lambda M_1 : \|L_1\|, \cdots \lambda M_t : \|L_m\|, \\
\lambda m_1 : \|M_1\|, \cdots \lambda m_t : \|M_t\|. \\
c_2 \ y_1 \cdots y_k \ b \ m_1 \cdots m_{h-1} \\
(\lambda q : \|u[s]_p \simeq v\|). \\
c_1 \ x_1 \cdots x_n \ b \ l_1 \cdots l_{i-1} \\
(\lambda r : \|s \simeq t\|, p \ (r (\lambda z. \simeq \ |u[z]_p| \ |v|) \ q)) \\
l_i \cdots l_m \\
m_h \cdots m_t
\end{array}
\]

Negative Replacement is almost the same, except that $p$ has type $\|u[t]_p \not\simeq v\|$, instead of $\|u[t]_p \simeq v\|$ and $q$ has type $\|u[s]_p \simeq v\|$ instead of $\|u[s]_p \not\simeq v\|$, so that $p \ (r (\lambda z. \simeq \ |u[z]_p| \ |v|) \ q)$ has type $\text{proof } \#b$.

Negative Replacement

\[
\begin{array}{c}
L_1; \cdots; L_{i-1}; s \not\simeq t; L_i; \cdots; L_m \rightarrow M_1; \cdots; M_{h-1}; u[s]_p \not\simeq v; M_h; \cdots; M_t \\
\end{array}
\]

\[
\begin{array}{c}
c_1 : \Pi x_1 : t. \cdots \Pi x_n : t. \Pi b : o. \ \Pi L_1 : b \rightarrow \cdots \rightarrow \|s \not\simeq t\| \rightarrow \cdots \rightarrow \|L_m\| \rightarrow \text{proof } \#b \\
c_2 : \Pi y_1 : t. \cdots \Pi y_n : t. \Pi b : o. \ \Pi M_1 : b \rightarrow \cdots \rightarrow \|u[s]_p \not\simeq v\| \rightarrow \cdots \rightarrow \|M_t\| \rightarrow \text{proof } \#b \\
d : \Pi z_1 : t. \cdots \Pi z_j : t. \Pi b : o. \ \Pi u[t]_p \not\simeq v \rightarrow \|L_1\| \rightarrow \cdots \rightarrow \|L_m\| \rightarrow \|M_1\| \rightarrow \cdots \rightarrow \|M_t\| \rightarrow \text{proof } \#b \\
d \rightarrow \lambda z_1 : t. \cdots \lambda z_j : t. \lambda b : o. \ \lambda p : \|u[t]_p \not\simeq v\|. \ \lambda M_1 : \|L_1\|, \cdots \lambda M_t : \|L_m\|, \\
\lambda m_1 : \|M_1\|, \cdots \lambda m_t : \|M_t\|. \\
c_2 \ y_1 \cdots y_k \ b \ m_1 \cdots m_{h-1} \\
(\lambda q : (\|u[s]_p \not\simeq v\|) \rightarrow \text{proof } \#b). \\
c_1 \ x_1 \cdots x_n \ b \ l_1 \cdots l_{i-1} \\
(\lambda r : \|s \not\simeq t\|, p \ (r (\lambda z. \simeq \ |u[z]_p| \ |v|) \ q)) \\
l_i \cdots l_m \\
m_h \cdots m_t
\end{array}
\]
We first declare the three input clauses:

1. \( c \simeq g(a); X \simeq f(b, Y) \)
2. \( g(X) \simeq f(Z, X) \)
3. \( g(c) \not\simeq g(f(X, Y)) \)
4. \( c \simeq f(Z, a); X \simeq f(b, Y) \)  \( \text{applied Positive Superposition on 2 and 1} \)
5. \( f(Z, a) \not\simeq f(b, Y); c \simeq f(b, Y) \)  \( \text{applied Equality Factoring on 4} \)
6. \( c \simeq f(b, a) \)  \( \text{applied Equality Resolution on 5} \)
7. \( g(f(b, a)) \not\simeq g(f(X, Y)) \)  \( \text{applied Negative Superposition on 6 and 3} \)
8. \( \square \)  \( \text{applied Equality Resolution on 7} \)

If we decompose the instantiations from the inferences, we get

1. \( c \simeq g(a); X \simeq f(b, Y) \)
2. \( g(X) \simeq f(Z, X) \)
3. \( g(c) \not\simeq g(f(X, Y)) \)
4. \( g(a) \simeq f(Z, a) \)  \( \text{applied Instantiation on 2 with } \sigma = \{ X \mapsto a \} \)
5. \( c \simeq f(Z, a); X \simeq f(b, Y) \)  \( \text{applied Positive Replacement on 4 and 1} \)
6. \( c \simeq f(Z, a); c \simeq f(b, Y) \)  \( \text{applied Instantiation on 5 with } \sigma = \{ X \mapsto c \} \)
7. \( f(Z, a) \not\simeq f(b, Y); c \simeq f(b, Y) \)  \( \text{applied Identical Equality Factoring on 6} \)
8. \( f(b, a) \not\simeq f(b, a); c \simeq f(b, a) \)  \( \text{applied Instantiation on 7 with } \sigma = \{ Y \mapsto a; Z \mapsto b \} \)
9. \( c \simeq f(b, a) \)  \( \text{applied Identical Equality Resolution on 8} \)
10. \( g(f(b, a)) \not\simeq g(f(X, Y)) \)  \( \text{applied Negative Replacement on 9 and 3} \)
11. \( g(f(b, a)) \not\simeq g(f(b, a)) \)  \( \text{applied Instantiation on 10 with } \sigma = \{ X \mapsto b; Y \mapsto a \} \)
12. \( \square \)  \( \text{applied Identical Equality Resolution on 11} \)

We have a unary function symbol \( g \), a binary function symbol \( f \) and three constants \( a \), \( b \) and \( c \). The context of the translation in the \( \lambda \)-calculus modulo is therefore:

\[
\begin{align*}
t &: \text{Type} \\
o &: \text{Type} \\
\doteq &: t \rightarrow t \rightarrow o \\
\doteq &: t \rightarrow t \rightarrow \text{Type} \\
\doteq &: t \rightarrow t \rightarrow \lambda x : t. \lambda y : t. \Pi p : (t \rightarrow o). \text{proof} (p \ x) \rightarrow \text{proof} (p \ y) \\
\doteq &: o \rightarrow o \rightarrow o \\
\text{proof} (\doteq x \ y) &: \doteq x \ y \\
\text{proof} (\doteq A \ B) &: \text{proof} A \rightarrow \text{proof} B \\
\text{refl} &: \Pi x : t. \doteq x \ x \\
\text{refl} &: \lambda x : t. \lambda p : t \rightarrow o. \lambda t : \text{proof} (p \ x). t \\
g &: t \rightarrow t \\
f &: t \rightarrow t \rightarrow t \\
a &: t \\
b &: t \\
c &: t \\
\end{align*}
\]

We first declare the three input clauses:

\[
c1 : \Pi X : t. \Pi Y : t. \Pi b : o. (\doteq c (g a) \rightarrow \text{proof} b) \rightarrow (\doteq X (f b Y) \rightarrow \text{proof} b) \rightarrow \text{proof} b
\]
A Shallow Embedding of Resolution and Superposition Proofs into the \( \lambda \)-Calculus Modulo Burel

We then declare the inferred clauses and define them as explained above:

\[\begin{align*}
c4 & : \Pi X : t. \Pi b : o. (\sim (g \ a) (f \ Z \ a) \rightarrow \text{proof } b) \rightarrow \text{proof } b \\
c5 & : \Pi X : t. \Pi Y : t. \Pi b : o. (\sim c (f \ Z \ a) \rightarrow \text{proof } b) \rightarrow (\sim X (f \ b \ Y) \rightarrow \text{proof } b) \rightarrow \text{proof } b \\
c6 & : \Pi X : t. \Pi Y : t. \Pi b : o. (\sim c (f \ Z \ a) \rightarrow \text{proof } b) \rightarrow (\sim c (f \ b \ Y) \rightarrow \text{proof } b) \rightarrow \text{proof } b \\
c7 & : \Pi X : t. \Pi Z : t. \Pi b : o. (\sim (f \ Z \ a) (f \ b \ Y) \rightarrow \text{proof } b) \rightarrow \text{proof } b \\
c8 & : \Pi a \rightarrow \text{proof } b) \rightarrow (\sim c (f \ b \ a) \rightarrow \text{proof } b) \rightarrow \text{proof } b \\
c9 & : \Pi b : o. (\sim c (f \ b \ a) \rightarrow \text{proof } b) \\
c10 & : \Pi X : t. \Pi Y : t. \Pi b : o. (\sim (g \ b \ a) (g \ (f \ X \ Y)) \rightarrow \text{proof } b) \rightarrow \text{proof } b \\
c11 & : \Pi b : o. (\sim (g \ (f \ b \ a)) (g \ (f \ b \ a)) \rightarrow \text{proof } b) \rightarrow \text{proof } b \\
c12 & : \Pi b : o. \text{proof } b \\
\end{align*}\]

### 3.3 Resolution Proofs Are Constructive Proofs

In the translation of resolution and superposition proofs above, we do not need any axiom for classical logic, which means that we have an intuitionistic proof. Furthermore, since the translation of a clause is intuitionistically implied by the translation of its corresponding formula, that means that the proof of unsatisfiability of a set of clauses by the resolution method is intuitionistic. However, the resolution method is in general used to refute the negation of a formula: to prove \( \neg A \), one proves that the clausal normal form of \( \neg A \) is unsatisfiable. To go to the proof of unsatisfiability of \( \neg A \) to a proof of \( A \), one needs a classical axiom (even without considering the clausification of \( \neg A \)).

This remark about constructiveness of resolution proofs is not so surprising. Indeed, given the clauses \( C_1, \ldots, C_m \) with correspond formulas \( A_1, \ldots, A_m \), proving the unsatisfiability of \( C_1, \ldots, C_m \) amounts to proving the sequent \( A_1, \ldots, A_m \vdash \) in the sequent calculus. But for this particular class of sequents, intuitionistic and classical logics coincide. Indeed, since there
are only atomic formulas under negations, and there are no implications, there can only be atomic formulas in the right-hand side of sequents in a proof of $A_1, \ldots, A_m \vdash$. Since only one of them can be used in each axiom rule closing a branch of the proof, we can restrict ourselves to sequents containing at most one formula in the right-hand side, as in the intuitionistic fragment.

4 Implementation in iProver Modulo

We have successfully implemented the technique above in iProver Modulo. iProver\[13\] is a prover for first-order logic based on the combination of two proof-search methods, namely instantiation-generation and resolution. iProver Modulo\[6\] is a patch to iProver to integrate Polarized Resolution Modulo\[9\] in it. iProver Modulo is available at \url{http://www.ensiie.fr/~guillaume.burel/blackandwhite_iProverModulo.html.en}. When iProver Modulo finds a pure resolution proof (for instance, when the instantiation-generation method is switched off), we are able to translate it to the \(\lambda\Pi\)-calculus modulo using the technique presented in this paper.

As said above, resolution- and superposition-based provers do not only use the inference rules presented above, but also use simplification rules that can be used to replace a set of clause by another. For instance, iProver Modulo uses

\[
\frac{L; C \quad \sigma(L); \sigma(C); D}{L; C \quad \sigma(C); D} \quad \text{Subsumption Resolution}
\]

where $P = \neg P$ and $\overline{P} = P$. After the simplification is performed, $\sigma(L); \sigma(C); D$ is no longer in the working space of the prover to search for a proof, but it can be used to translate a proof once it has been found. It is possible to infer the clause $\sigma(C); D$ but using the derivation

\[
\frac{L; C \quad \sigma(L); \sigma(C)}{\sigma(L); \sigma(C); D} \quad \text{Instantiation}
\]

\[
\frac{\sigma(L); \sigma(C) \quad \sigma(L); \sigma(C) \quad \sigma(C); D}{\sigma(C); D} \quad \text{Identical Resolution}
\]

\[
\frac{\sigma(L); \sigma(C) \quad \sigma(L); \sigma(C) \quad \sigma(C); D}{\sigma(C); D} \quad \text{Identical Factoring}
\]

and this derivation can be translated as usual.

In practice, when iProver Modulo is run with the option --dedukti-out-proof true, if a proof using only the resolution method is found, it is output in Dedukti's syntax (in which $\lambda x : a. t$ and $\Pi x : a. b$ are respectively written $x : a \to t$ and $x : a \to b$) and can be checked by the dkparse tool. For instance, for the unsatisfiability of the two clauses in Example 1, iProver Modulo outputs:

```plaintext
o : Type.
proof : (o -> Type).
i : Type.
a : i.
b : i.
p : (i -> (i -> Type)).
clause3 : (X1 : i -> (X0 : i -> (bot_var : o -> ((p X0 a -> proof bot_var) -> ((p X0 X1 -> proof bot_var) -> proof bot_var))))).
clause2 : (X0 : i -> (bot_var : o -> ((p X0 a -> proof bot_var) -> proof bot_var)))).
[] clause2 --> (X0 : i => (bot_var : o => ((lit1 : (p X0 a -> proof bot_var) => proof bot_var) => clause3 a X0 bot_var lit1 lit1))).
clause4 : (X0 : i -> (bot_var : o -> (((p b X0 -> proof bot_var) -> proof bot_var) => proof bot_var)
```
A Shallow Embedding of Resolution and Superposition Proofs into the \( \lambda \Pi \)-Calculus Modulo Burel

\[
\rightarrow \text{proof } \text{bot_var}))).
\]

\[
\text{clause1} : (\text{bot_var} : o \rightarrow \text{proof } \text{bot_var}).
\]

\[
[] \text{clause1} \rightarrow (\text{bot_var} : o \Rightarrow
\text{clause2} b \text{ bot_var} (tp : p b a \Rightarrow
\text{clause4} a \text{ bot_var} (tnp : (p b a \rightarrow \text{proof } \text{bot_var}) \Rightarrow tnp tp))).
\]

The input clauses are \text{clause3} and \text{clause4}, and the false formula \( \Pi \flat : o. \text{proof } \flat \) is proved by \text{clause1}. Note that contrarily to what is detailed above to ease the comprehension, instantiations are integrated in the inference rules: \text{clause2} is inferred from \text{clause3} by Factoring (with \( \sigma = \{X1 \mapsto a\} \)), and \text{clause1} from \text{clause2} and \text{clause4} by Resolution (with \( \sigma \) mapping the \( X0 \) of \text{clause2} to \( b \) and the \( X0 \) of \text{clause4} to \( a \)).

Conclusion

We have presented a shallow embedding of the proofs found by state-of-art first-order automated theorem provers into the \( \lambda \Pi \)-calculus. We also have described its implementation in \text{iProver Modulo}. This work is a first step towards the interoperability of automated theorem provers and proof assistants. We can now envisage to combine proofs coming from Coq, HOL, and \text{iProver modulo}, by linking their translations in Dedukti. To that purpose, as explained in the introduction, the fact that the embeddings are shallow will be extremely useful. We now consider further works.

An implementation of the translation of superposition proofs would let us see if such an embedding can really be used in practice. A good candidate for integrating this translation is Zipperposition, a first-order theorem prover based on superposition, written in OCaml and developed as a experimental platform to test ideas around the superposition calculus. Zipperposition is available at [https://www.rocq.inria.fr/deducteam/Zipperposition/](https://www.rocq.inria.fr/deducteam/Zipperposition/).

Moreover, first-order theorem provers generally do not take as inputs only set of clauses to be proved unsatisfiable, but they also can handle full first-order formulas. To be able to translate these proofs into the \( \lambda \Pi \)-calculus modulo, we should be able to express in the \( \lambda \Pi \)-calculus modulo the transformation of formulas into clausal normal form. This raises two issues: first, some transformations need classical logic. To handle them, a possibility is to add a classical axiom, for instance \( \text{nnpp} : \Pi p : o. \Pi \flat : o. ((\text{proof } p \rightarrow \text{proof } \flat) \rightarrow \text{proof } \flat) \rightarrow \text{proof } p \). A more difficult point is that for some transformations, the resulting set of formulas is not logically equivalent to the first one, but is only equisatisfiable. This is the case for instance of the elimination of an existential quantifier using a Skolem symbol. To solve this, one should probably transform the proof back to reintegrate the existential variables introduced by Skolemization.

Another interesting idea would be to use \text{iProver Modulo} to output a Dedukti proof for each inference step of a proof found by another prover, as could be described in the TSTP format [18]. Then, by recombining each of these steps, we would obtain a whole proof of the original formula, at least if only inference rules that are really logical implications are used. Nevertheless, this is not immediate, because for the moment we only translate proofs of unsatisfiability of set of clauses, and the combination of such proofs would require to link clauses with the clausal normal form of their negation: a proof that \( C1 \) and \( C2 \) leads to \( C3 \) will indeed be a proof that \( C1, C2 \) and the clausal normal form of \( \neg C3 \) is unsatisfiable.
A Shallow Embedding of Resolution and Superposition Proofs into the $\lambda\Pi$-Calculus Modulo

References