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

Guillaume Burel
ÉNSIIE/Cédric
1 square de la résistance, 91025 Évry cedex, France
guillaume.burel@ensiie.fr
http://www.ensiie.fr/~guillaume.burel/

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]. Nervertheless, 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.

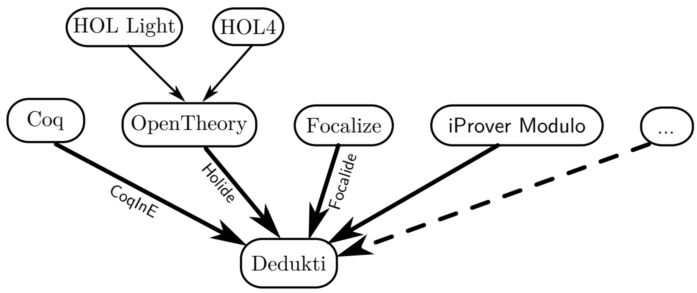

Figure 1: Dedukti as a universal proof language

a deep embedding by the fact that it reuse 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 $||\cdot||_A$ and $||\cdot||_B$ 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 [7, 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, λP, 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 dkparse, was implemented. dkparse is available at `https://www.rocq.inria.fr/deducteam/Dedukti/` . Tools related to Dedukti also include a translator of Coq proofs to Dedukti, namely CoqInE [4, `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, `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 [`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 [16]. This includes for instance the provers Vampire [15], SPASS [19] 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.

$$\textsf{Empty}\ \frac{}{\emptyset\ \mathrm{WF}} \qquad \textsf{Declaration}\ \frac{\Gamma\ \mathrm{WF} \qquad \Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A\ \mathrm{WF}}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$$\textsf{Sort}\ \frac{\Gamma\ \mathrm{WF}}{\Gamma \vdash \mathsf{Type} : \mathsf{Kind}} \qquad \textsf{Variable}\ \frac{\Gamma\ \mathrm{WF} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsf{Product}\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma[x : A] \vdash B : s}{\Gamma \vdash \Pi x : A.\ B : s}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$$\textsf{Application}\ \frac{\Gamma \vdash T : \Pi x : A.\ B \qquad \Gamma \vdash U : A}{\Gamma \vdash (T\ U) : \{U/x\}B}$$

$$\textsf{Abstraction}\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : s \qquad \Gamma, x : A \vdash T : B}{\Gamma \vdash \lambda x : A.\ T : \Pi x : A.\ B}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$$\textsf{Conversion}\ \frac{\Gamma \vdash T : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash T : B}\ s \in \{\mathsf{Type}, \mathsf{Kind}\} \text{ and } A \equiv_\beta B$$

Figure 2: Type System for the $\lambda\Pi$-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 $\lambda\Pi$-calculus modulo. In Section 2, we describe the shallow embedding of first-order logic with equality in the $\lambda\Pi$-calculus modulo. Section 3 details the translation of resolution and superposition proofs. Its implementation in iProver Modulo is outlined in Section 4.

1 The $\lambda\Pi$-Calculus Modulo

The $\lambda\Pi$-calculus modulo [7, 5] is an extension of the $\lambda\Pi$-calculus, a proof language for minimal first-order logic also known as LF, λP, etc [11]. The $\lambda\Pi$-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 $\lambda\Pi$-calculus are defined by the grammar

$$M, N, A, B ::= x \mid \lambda x : A.\ M \mid \Pi x : A.\ B \mid M\ N \mid \mathsf{Type} \mid \mathsf{Kind}$$

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 "Γ WF" means that a context Γ is well-formed, and the judgment $\Gamma \vdash T : A$ must be read as "T has type A in the context Γ".

In the Conversion rule of the $\lambda\Pi$-calculus, $A \equiv_\beta B$ means that A and B are β-convertible. In the $\lambda\Pi$-calculus *modulo*, this conversion rule is extended by means of well-typed rewriting rules:

Definition 1 (Rewrite rule)**.** A rewrite rule is a quadruple $\Delta :\cdot\ l \hookrightarrow^A r$ composed of a context Δ and three terms l, r and A. It is well typed in a context Γ if:

- the context Γ, Δ is well-formed;

- $\Gamma, \Delta \vdash l : A$ and $\Gamma, \Delta \vdash r : A$ are derivable judgments.

Intuitively, Δ 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, Δ and A can often be inferred from l and r, in which case we will omit them and simply write $l \hookrightarrow r$. As usual, a term t is rewritten by a rewrite rule $l \hookrightarrow r$ to a term s if their exists a substitution θ such that a subterm of t at a position $\mathfrak{p}$ is equal to $\theta(l)$ and s is equal to t where the subterm at position $\mathfrak{p}$ 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:

$$\mathsf{Rewrite}\ \frac{\Gamma, \Delta \vdash l : A \qquad \Gamma, \Delta \vdash r : A}{\Gamma, (\Delta :\cdot\ l \hookrightarrow^{A} r)\ \mathrm{WF}}$$

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

$$\mathsf{Conversion}\ \frac{\Gamma \vdash T : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash T : B}\ s \in \{\mathsf{Type}, \mathsf{Kind}\} \text{ and } A \equiv_\Gamma B$$

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 => t` and `x : a -> b`, and a rewriting rule $\Delta :\cdot\ l \hookrightarrow r$ is declared as `[Δ] l --> r.` The tool 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 λΠ-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 $\lambda\Pi$-calculus modulo [7].

We use standard definitions for terms, predicates, first-order propositions (with connectives $\bot, \neg, \Rightarrow, \wedge, \vee$ 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 proof that is defined by means of rewriting rules.

To define the deep embedding, we first define two constants ι and o of type Type that contains respectively the translation of terms and propositions. We add constants $\dot\bot : o$, $\dot\neg : o \to o$, $\dot\Rightarrow : o \to o \to o$, $\dot\vee : o \to o \to o$, $\dot\wedge : o \to o \to o$, $\dot\forall : (\iota \to o) \to o$, $\dot\exists : (\iota \to o) \to o$ for the translation of connectives and quantifiers. For each function symbol f of arity n we add a constant $f : \underbrace{\iota \to \cdots \to \iota \to}_{n \text{ times}} \iota$, and for each predicate symbol p of arity n we add a constant $\dot p : \underbrace{\iota \to \cdots \to \iota \to}_{n \text{ times}} o$. 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:

$$\begin{aligned}
|X| &= X && \text{if } X \text{ is a variable}\\
|f(t1,\dots,t_n)| &= f\ |t_1|\ \cdots\ |t_n|\\
|p(t1,\dots,t_n)| &= \dot{p}\ |t_1|\ \cdots\ |t_n|\\
|\bot| &= \dot{\bot}\\
|\neg A| &= \dot{\neg}\ |A|\\
|A \Rightarrow B| &= \dot{\Rightarrow}\ |A|\ |B|\\
|A \vee B| &= \dot{\vee}\ |A|\ |B|\\
|A \wedge B| &= \dot{\wedge}\ |A|\ |B|\\
|\forall X.A| &= \dot{\forall}\ (\lambda X : \iota.\ |A|)\\
|\exists X.A| &= \dot{\exists}\ (\lambda X : \iota.\ |A|)
\end{aligned}$$

The shallow embedding is defined by $||A|| = \mathsf{proof}\ |A|$ where proof is a decoding function of type $o \to \mathsf{Type}$. 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 λΠ-calculus modulo. $\dot{\Rightarrow}$ is for instance related with $\to$, $\dot{\forall}$ with Π, whereas the other connectives are related with their impredicative encoding in λΠ. We can add a constant p of type $\underbrace{\iota \to \cdots \to \iota \to}_{n \text{ times}} \mathsf{Type}$ to get a shallow embedding of each predicate symbol p whose arity is n. The rules defining proof are therefore:

$$\begin{aligned}
\mathsf{proof}\ (\dot{p}\ t_1\ \cdots\ t_n) &\hookrightarrow p\ t_1\ \cdots\ t_n\\
\mathsf{proof}\ \dot{\bot} &\hookrightarrow \Pi\flat : o.\ \mathsf{proof}\ \flat\\
\mathsf{proof}\ (\dot{\neg}\ A) &\hookrightarrow \Pi\flat : o.\ \mathsf{proof}\ A \to \mathsf{proof}\ \flat\\
\mathsf{proof}\ (\dot{\Rightarrow}\ A\ B) &\hookrightarrow \mathsf{proof}\ A \to \mathsf{proof}\ B\\
\mathsf{proof}\ (\dot{\vee}\ A\ B) &\hookrightarrow \Pi\flat : o.\ (\mathsf{proof} A \to \mathsf{proof}\ \flat) \to (\mathsf{proof} B \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat\\
\mathsf{proof}\ (\dot{\wedge}\ A\ B) &\hookrightarrow \Pi\flat : o.\ (\mathsf{proof} A \to \mathsf{proof} B \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat\\
\mathsf{proof}\ (\dot{\forall}\ f) &\hookrightarrow \Pi X : \iota.\ \mathsf{proof}\ (f\ X)\\
\mathsf{proof}\ (\dot{\exists}\ f) &\hookrightarrow \Pi\flat : o.\ (\Pi X : \iota.\ \mathsf{proof}\ (f\ X) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat
\end{aligned}$$

where $\flat$ 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 λΠ-calculus modulo with 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 $L_1; \cdots ; L_m$. It corresponds to the formula $\forall X_1.\ \dots\forall X_n.\ L_1 \vee \cdots \vee L_m$ where $X_1, \dots, X_n$ are the free variables of $L_1, \dots, L_m$. 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

$$||L_1; \cdots ; L_m|| = \Pi X_1 : \iota.\ \dots \Pi X_n : \iota.\ \Pi\flat : o.\ [\![L_1]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \mathsf{proof}\ \flat$$

where $X_1, \ldots, X_n$ are the free variables in the clause and $[\![P]\!]_\flat = ||P|| \to \mathsf{proof}\ \flat$ for a positive literal P and $[\![\neg P]\!]_\flat = (||P|| \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$ for a negative literal $\neg P$. The empty clause is therefore translated as $\Pi\flat : o.\ \mathsf{proof}\ \flat$, 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 $\forall X_1.\ \ldots \forall X_n.\ L_1 \vee \cdots \vee L_m$. 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 λΠ-calculus modulo, it is possible to *define* the equality predicate using Leibniz law.

$$\begin{aligned} &\simeq : \iota \to \iota \to \mathsf{Type} \\ &\simeq \hookrightarrow \lambda x : \iota.\ \lambda y : \iota.\ \Pi p : (\iota \to o).\ \mathsf{proof}\ (p\ x) \to \mathsf{proof}\ (p\ y) \end{aligned}$$

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

$$\begin{aligned} &\mathsf{refl} : \Pi x : \iota.\ \simeq\ x\ x \\ &\mathsf{refl} \hookrightarrow \lambda x : \iota.\ \lambda p : \iota \to o.\ \lambda t : \mathsf{proof}\ (p\ x).\ t \end{aligned}$$

Commutativity has the following proof:

$$\begin{aligned} &\mathsf{comm} : \Pi x : \iota.\ \Pi y : \iota.\ \simeq\ x\ y \to \simeq\ y\ x \\ &\mathsf{comm} \hookrightarrow \lambda x : \iota.\ \lambda y : \iota.\ \lambda e : \simeq\ x\ y.\ \lambda p : \iota \to o.\ e\ (\lambda z : \iota.\ \Rightarrow (p\ z)\ (p\ x))\ (\lambda t : \mathsf{proof}\ (p\ x).\ t) \end{aligned}$$

3 Translating resolution and superposition proofs

3.1 Resolution

A derivation in resolution [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.

$$\text{Resolution}\ \frac{P; C \qquad \neg Q; D}{\sigma(C; D)}\ \sigma = mgu(P, Q) \qquad\qquad \text{Factoring}\ \frac{L; K; C}{\sigma(L; C)}\ \sigma = mgu(L, K)$$

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

$$\text{Instantiation}\ \frac{C}{\sigma(C)} \qquad \text{Identical Resolution}\ \frac{P; C \qquad \neg P; D}{C; D} \qquad \text{Identical Factoring}\ \frac{L; L; C}{L; C}$$

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 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 λΠ-calculus modulo, we declare a constant c_i of type $||C_i||$ for each $1 \le i \le n$, 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 $\forall \flat.\ \mathsf{proof}\ \flat$ 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 $\flat$ to be proved, m functions that given a term of type $||L_i||$ return a proof of $\flat$, and that return a proof of $\flat$.

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:

$$\textsf{Instantiation}\ \frac{L_1; \cdots ; L_m}{\sigma(L_1); \cdots ; \sigma(L_m)}$$

$$\begin{aligned}
&c : \Pi x_1 : \iota.\ \ldots \Pi x_n : \iota.\ \Pi \flat : o.\ [\![L_1]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \mathsf{proof}\ \flat \\
&d : \Pi y_1 : \iota.\ \ldots \Pi y_k : \iota.\ \Pi \flat : o.\ [\![\sigma(L_1)]\!]_\flat \to \cdots \to [\![\sigma(L_m)]\!]_\flat \to \mathsf{proof}\ \flat \\
&d \hookrightarrow \lambda y_1 : \iota.\ \ldots \lambda y_k : \iota.\ c\ (\sigma(x_1))\ \cdots\ (\sigma(x_n))
\end{aligned}$$

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

$$\textsf{Identical Factoring}\ \frac{L_1; \cdots ; L_i; L_i; \cdots ; L_m}{L_1; \cdots ; L_i; \cdots ; L_m}$$

$$\begin{aligned}
&c : \Pi x_1 : \iota.\ \ldots \Pi x_n : \iota.\ \Pi \flat : o.\ [\![L_1]\!]_\flat \to \cdots \to [\![L_i]\!]_\flat \to [\![L_i]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \mathsf{proof}\ \flat \\
&d : \Pi x_1 : \iota.\ \ldots \Pi x_n : \iota.\ \Pi \flat : o.\ [\![L_1]\!]_\flat \to \cdots \to [\![L_i]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \mathsf{proof}\ \flat \\
&d \hookrightarrow \lambda x_1 : \iota.\ \ldots \lambda x_n : \iota.\ \lambda \flat : o.\ \lambda l_1 : [\![L_1]\!]_\flat.\ \cdots\ \lambda l_m : [\![L_m]\!]_\flat.\ c\ x_1\ \cdots\ x_n\ \flat\ l_1\ \cdots\ l_i\ l_i\ \cdots\ l_m
\end{aligned}$$

To translate a resolution step, we can use the atom P and its negation to get the proof of $\flat$. More precisely, we can use as term of type $[\![P]\!]_\flat = ||P|| \to \mathsf{proof}\ \flat$ 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 $M_1; \cdots ; \neg P \cdots ; M_l$ where the term for type $[\![\neg P]\!]_\flat = (||P|| \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$ is the function that take a term tnp of type $||P|| \to \mathsf{proof}\ \flat$ and return $tnp\ tp$, which is of type $\mathsf{proof}\ \flat$.

$$\textsf{Identical Resolution}\ \frac{L_1; \cdots ; L_{i-1}; P; L_i; \cdots ; L_m \qquad M_1; \cdots ; M_{h-1}; \neg P; M_h; \cdots ; M_l}{L_1; \cdots ; L_m; M_1; \cdots ; M_l}$$

$$\begin{aligned}
&c1 : \Pi x_1 : \iota.\ \ldots \Pi x_n : \iota.\ \Pi \flat : o.\ [\![L_1]\!]_\flat \to \cdots \to [\![P]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \mathsf{proof}\ \flat \\
&c2 : \Pi y_1 : \iota.\ \ldots \Pi y_k : \iota.\ \Pi \flat : o.\ [\![M_1]\!]_\flat \to \cdots \to [\![\neg P]\!]_\flat \to \cdots \to [\![M_l]\!]_\flat \to \mathsf{proof}\ \flat \\
&d : \Pi z_1 : \iota.\ \ldots \Pi z_j : \iota.\ \Pi \flat : o.\ [\![L_1]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to [\![M_1]\!]_\flat \to \cdots \to [\![M_l]\!]_\flat \to \mathsf{proof}\ \flat \\
&d \hookrightarrow \lambda z_1 : \iota.\ \ldots \lambda z_j : \iota.\ \lambda \flat : o.\ \lambda l_1 : [\![L_1]\!]_\flat.\ \cdots\ \lambda l_m : [\![L_m]\!]_\flat. \\
&\qquad\qquad \lambda m_1 : [\![M_1]\!]_\flat.\ \cdots\ \lambda m_l : [\![M_l]\!]_\flat. \\
&\qquad\qquad c1\ x_1\ \cdots\ x_n\ \flat\ l_1\ \cdots\ l_{i-1} \\
&\qquad\qquad\quad (\lambda tp : ||P||.\ c2\ y_1\ \cdots\ y_k\ \flat\ m_1\ \cdots\ m_{h-1} \\
&\qquad\qquad\qquad (\lambda tnp : (||P|| \to \mathsf{proof}\ \flat).\ tnp\ tp)\ m_h\ \cdots\ m_l) \\
&\qquad\qquad\quad l_i\ \cdots\ l_m
\end{aligned}$$

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 Factoring on 1
4	$\square$	applying Resolution on 2 and 3

If we decompose the instantiations 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 Instantiation on 1 with $\sigma = \{Y \mapsto a\}$
4	$p(X,a)$	applying Identical Factoring on 3
5	$p(b,a)$	applying Instantiation on 4 with $\sigma = \{X \mapsto b\}$
6	$\neg p(b,a)$	applying Instantiation on 2 with $\sigma = \{Y \mapsto a\}$
7	$\square$	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 λΠ-calculus modulo is therefore

$$\begin{aligned}
&\iota : \mathsf{Type}\\
&o : \mathsf{Type}\\
&\mathsf{proof} : o \to \mathsf{Type}\\
&\dot{p} : \iota \to \iota \to o\\
&p : \iota \to \iota \to \mathsf{Type}\\
&\mathsf{proof}\ (\dot{p}\ x\ y) \hookrightarrow p\ x\ y\\
&a : \iota\\
&b : \iota
\end{aligned}$$

We first declare the two input clauses:

$$\begin{aligned}
&c1 : \Pi X : \iota.\ \Pi Y : \iota.\ \Pi\flat : o.\ (p\ X\ Y \to \mathsf{proof}\ \flat) \to (p\ X\ a \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat\\
&c2 : \Pi Y : \iota.\ \Pi\flat : o.\ ((p\ b\ Y \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat
\end{aligned}$$

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

$$\begin{aligned}
&c3 : \Pi X : \iota.\ \Pi\flat : o.\ (p\ X\ a \to \mathsf{proof}\ \flat) \to (p\ X\ a \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat\\
&c3 \hookrightarrow \lambda X : \iota.\ c1\ X\ a\\
&c4 : \Pi X : \iota.\ \Pi\flat : o.\ (p\ X\ a \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat\\
&c4 \hookrightarrow \lambda X : \iota.\ \lambda\flat : o.\ \lambda l : (p\ X\ a \to \mathsf{proof}\ \flat).\ c3\ X\ \flat\ l\ l\\
&c5 : \Pi\flat : o.\ (p\ b\ a \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat\\
&c5 \hookrightarrow c4\ b\\
&c6 : \Pi\flat : o.\ ((p\ b\ a \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat\\
&c6 \hookrightarrow c2\ a\\
&c7 : \Pi\flat : o.\ \mathsf{proof}\ \flat\\
&c7 \hookrightarrow \lambda\flat : o.\ c5\ \flat\ (\lambda tp : p\ b\ a.\ c6\ \flat\ (\lambda tnp : p\ b\ a \to \mathsf{proof}\ \flat.\ tnp\ tp))
\end{aligned}$$

3.2 Superposition

Superposition can be seen as an extension of resolution to handle equality better. Superposition primarily uses four inference rules ($u \not\simeq v$ denotes $\neg(u \simeq v)$):

$$\textsf{Equality Resolution}\ \frac{u \not\simeq v; R}{\sigma(R)}\ \sigma = mgu(u, v) \quad \textsf{Negative Superposition}\ \frac{s \simeq t; S \qquad u \not\simeq v; R}{\sigma(u[t]_{\mathfrak{p}} \not\simeq v; S; R)}\ \sigma = mgu(u_{|\mathfrak{p}}, s)$$

$$\textsf{Positive Superposition}\ \frac{s \simeq t; S \qquad u \simeq v; R}{\sigma(u[t]_{\mathfrak{p}} \simeq v; S; R)}\ \sigma = mgu(u_{|\mathfrak{p}}, s) \quad \textsf{Equality Factoring}\ \frac{s \simeq t; u \simeq v; R}{\sigma(t \not\simeq v; u \simeq v; R)}\ \sigma = mgu(s, u)$$

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:

$$\textsf{Identical Equality Resolution}\ \frac{u \not\simeq u; R}{R} \qquad \textsf{Negative Replacement}\ \frac{s \simeq t; S \qquad u[s]_{\mathfrak{p}} \not\simeq v; R}{u[t]_{\mathfrak{p}} \not\simeq v; S; R}$$

$$\textsf{Positive Replacement}\ \frac{s \simeq t; S \qquad u[s]_{\mathfrak{p}} \simeq v; R}{u[t]_{\mathfrak{p}} \simeq v; S; R} \qquad \textsf{Identical Equality Factoring}\ \frac{s \simeq t; s \simeq v; R}{t \not\simeq v; s \simeq v; R}$$

Once more, these rules can be applied modulo commutativity of ; and $\simeq$. For $\simeq$, it can be taken into account using the $\textsf{comm}$ 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, $\textsf{Identical Equality Resolution}$ is rather easy to translate. Indeed, a term of type $[\![u \not\simeq u]\!]_\flat = (\simeq\ u\ u \to \textsf{proof}\ \flat) \to \textsf{proof}\ \flat$ can be $\lambda p : (||u \simeq u|| \to \textsf{proof}\ \flat).\ p\ (\textsf{refl}\ u)$.

$$\textsf{Identical Equality Resolution}\ \frac{L_1; \cdots ; L_{i-1}; u \not\simeq u; L_i \cdots ; L_m}{L_1; \cdots ; L_m}$$

$$\begin{aligned}
&c : \Pi x_1 : \iota.\ \ldots \Pi x_n : \iota.\ \Pi\flat : o.\ [\![L_1]\!]_\flat \to \cdots \to [\![u \not\simeq u]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \textsf{proof}\ \flat \\
&d : \Pi y_1 : \iota.\ \ldots \Pi y_k : \iota.\ \Pi\flat : o.\ [\![L_1]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \textsf{proof}\ \flat \\
&d \hookrightarrow\ \lambda y_1 : \iota.\ \ldots \lambda y_k : \iota.\ \lambda\flat : o.\ \lambda l_1 : [\![L_1]\!]_\flat.\ \cdots\ \lambda l_m : [\![L_m]\!]_\flat. \\
&\qquad\qquad c\ x_1\ \ldots\ x_n\ \flat\ l_1\ \ldots\ l_{i-1}\ (\lambda p : (||u \simeq u|| \to \textsf{proof}\ \flat).\ p\ (\textsf{refl}\ u))\ l_i\ \ldots\ l_m
\end{aligned}$$

For $\textsf{Identical Equality Factoring}$, we somehow need to refute $s \simeq t$ from $s \simeq v$ and $t \not\simeq v$. If we consider a term p of type $[\![t \not\simeq v]\!]_\flat = (\simeq\ t\ v \to \textsf{proof}\ \flat) \to \textsf{proof}\ \flat$, a term q of type $[\![s \simeq v]\!]_\flat = \simeq\ s\ v \to \textsf{proof}\ \flat$ and a term r of type $||s \simeq t|| = \simeq\ s\ t$, the term $p\ (r\ (\lambda z : \iota. \dot{\Rightarrow} (\dot{\simeq}\ z\ v)\ \flat)\ q)$ has type $\textsf{proof}\ \flat$.

$$\textsf{Identical Equality Factoring}\ \frac{L_1; \cdots ; L_{h-1}; s \simeq t; L_h; \cdots ; L_{i-1}; s \simeq v; L_i; \cdots ; L_m}{t \not\simeq v; s \simeq v; L_1; \cdots ; L_m}$$

$$\begin{aligned}
&c : \Pi x_1 : \iota.\ \ldots \Pi x_n : \iota.\ \Pi\flat : o.\ [\![L_1]\!]_\flat \to \cdots \to [\![s \simeq t]\!]_\flat \to \cdots \to [\![s \simeq v]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \textsf{proof}\ \flat \\
&d : \Pi y_1 : \iota.\ \ldots \Pi y_k : \iota.\ \Pi\flat : o.\ [\![t \not\simeq v]\!]_\flat \to [\![s \simeq v]\!]_\flat \to [\![L_1]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \textsf{proof}\ \flat
\end{aligned}$$

$$d \hookrightarrow \lambda y_1 : \iota.\ \dots \lambda y_k : \iota.\ \lambda \flat : o.\ \lambda p : [\![t \not\simeq v]\!]_\flat.\ \lambda q : [\![s \simeq v]\!]_\flat.\ \lambda l_1 : [\![L_1]\!]_\flat.\ \cdots\ \lambda l_m : [\![L_m]\!]_\flat.$$
$$c\ x_1\ \dots\ x_n\ \flat\ l_1\ \dots\ l_{h-1}\ (\lambda r : ||s \simeq t||.\ p\ (r\ (\lambda z : \iota.\ \Rightarrow (\dot{\simeq}\ z\ v)\ \flat)\ q))\ l_h\ \dots\ l_{i-1}\ q\ l_i\ \dots\ l_m$$

For Positive Replacement, we can use the following idea: given a term p of type $[\![u[t]_\mathfrak{p} \simeq v]\!]_\flat = ||u[t]_\mathfrak{p} \simeq v|| \to \mathsf{proof}\ \flat$, a term q of type $||u[s]_\mathfrak{p} \simeq v||$ and a term r of type $||s \simeq t||$, the term $p\ (r\ (\lambda z.\ \dot{\simeq}\ |u[z]_\mathfrak{p}|\ |v|)\ q)$ has type $\mathsf{proof}\ \flat$.

$$\textsf{Positive Replacement}\ \frac{L_1; \cdots; L_{i-1}; s \simeq t; L_i; \cdots; L_m \qquad M_1; \cdots; M_{h-1}; u[s]_\mathfrak{p} \simeq v; M_h; \cdots; M_l}{u[t]_\mathfrak{p} \simeq v; L_1; \cdots; L_m; M_1; \cdots; M_l}$$

$$c1 : \Pi x_1 : \iota.\ \dots \Pi x_n : \iota.\ \Pi \flat : o.\ [\![L_1]\!]_\flat \to \cdots \to [\![s \simeq t]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \mathsf{proof}\ \flat$$
$$c2 : \Pi y_1 : \iota.\ \dots \Pi y_k : \iota.\ \Pi \flat : o.\ [\![M_1]\!]_\flat \to \cdots \to [\![u[s]_\mathfrak{p} \simeq v]\!]_\flat \to \cdots \to [\![M_l]\!]_\flat \to \mathsf{proof}\ \flat$$
$$d : \Pi z_1 : \iota.\ \dots \Pi z_j : \iota.\ \Pi \flat : o.\ [\![u[t]_\mathfrak{p} \simeq v]\!]_\flat \to [\![L_1]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to [\![M_1]\!]_\flat \to \cdots \to [\![M_l]\!]_\flat \to \mathsf{proof}\ \flat$$
$$\begin{aligned} d \hookrightarrow \lambda z_1 : \iota.\ \dots \lambda z_j : \iota.\ \lambda \flat : o.\ \lambda p : [\![u[t]_\mathfrak{p} \simeq v]\!]_\flat.\ & \lambda l_1 : [\![L_1]\!]_\flat.\ \cdots\ \lambda l_m : [\![L_m]\!]_\flat. \\ & \lambda m_1 : [\![M_1]\!]_\flat.\ \cdots\ \lambda m_l : [\![M_l]\!]_\flat. \\ & c2\ y_1\ \cdots\ y_k\ \flat\ m_1\ \cdots\ m_{h-1} \\ & \quad (\lambda q : ||u[s]_\mathfrak{p} \simeq v||. \\ & \qquad c1\ x_1\ \cdots\ x_n\ \flat\ l_1\ \cdots\ l_{i-1} \\ & \qquad\quad (\lambda r : ||s \simeq t||.\ p\ (r\ (\lambda z.\ \dot{\simeq}\ |u[z]_\mathfrak{p}|\ |v|)\ q)) \\ & \qquad\quad l_i\ \cdots\ l_m) \\ & \quad m_h\ \cdots\ m_l \end{aligned}$$

Negative Replacement is almost the same, except that p has type $[\![u[t]_\mathfrak{p} \not\simeq v]\!]_\flat$ instead of $[\![u[t]_\mathfrak{p} \simeq v]\!]_\flat$ and q has type $||u[s]_\mathfrak{p} \simeq v|| \to \mathsf{proof}\ \flat$ instead of $||u[s]_\mathfrak{p} \simeq v||$, so that $p\ (r\ (\lambda z.\ \Rightarrow (\dot{\simeq}\ |u[z]_\mathfrak{p}|\ |v|)\ \flat)\ q)$ has type $\mathsf{proof}\ \flat$.

$$\textsf{Negative Replacement}\ \frac{L_1; \cdots; L_{i-1}; s \simeq t; L_i; \cdots; L_m \qquad M_1; \cdots; M_{h-1}; u[s]_\mathfrak{p} \not\simeq v; M_h; \cdots; M_l}{u[t]_\mathfrak{p} \not\simeq v; L_1; \cdots; L_m; M_1; \cdots; M_l}$$

$$c1 : \Pi x_1 : \iota.\ \dots \Pi x_n : \iota.\ \Pi \flat : o.\ [\![L_1]\!]_\flat \to \cdots \to [\![s \simeq t]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to \mathsf{proof}\ \flat$$
$$c2 : \Pi y_1 : \iota.\ \dots \Pi y_k : \iota.\ \Pi \flat : o.\ [\![M_1]\!]_\flat \to \cdots \to [\![u[s]_\mathfrak{p} \not\simeq v]\!]_\flat \to \cdots \to [\![M_l]\!]_\flat \to \mathsf{proof}\ \flat$$
$$d : \Pi z_1 : \iota.\ \dots \Pi z_j : \iota.\ \Pi \flat : o.\ [\![u[t]_\mathfrak{p} \not\simeq v]\!]_\flat \to [\![L_1]\!]_\flat \to \cdots \to [\![L_m]\!]_\flat \to [\![M_1]\!]_\flat \to \cdots \to [\![M_l]\!]_\flat \to \mathsf{proof}\ \flat$$
$$\begin{aligned} d \hookrightarrow \lambda z_1 : \iota.\ \dots \lambda z_j : \iota.\ \lambda \flat : o.\ \lambda p : [\![u[t]_\mathfrak{p} \not\simeq v]\!]_\flat.\ & \lambda l_1 : [\![L_1]\!]_\flat.\ \cdots\ \lambda l_m : [\![L_m]\!]_\flat. \\ & \lambda m_1 : [\![M_1]\!]_\flat.\ \cdots\ \lambda m_l : [\![M_l]\!]_\flat. \\ & c2\ y_1\ \cdots\ y_k\ \flat\ m_1\ \cdots\ m_{h-1} \\ & \quad (\lambda q : (||u[s]_\mathfrak{p} \simeq v|| \to \mathsf{proof}\ \flat). \\ & \qquad c1\ x_1\ \cdots\ x_n\ \flat\ l_1\ \cdots\ l_{i-1} \\ & \qquad\quad (\lambda r : ||s \simeq t||.\ p\ (r\ (\lambda z.\ \Rightarrow (\dot{\simeq}\ |u[z]_\mathfrak{p}|\ |v|)\ \flat)\ q)) \\ & \qquad\quad l_i\ \cdots\ l_m) \\ & \quad m_h\ \cdots\ m_l \end{aligned}$$

Example 2. We want to refute the set of three clauses $c \simeq g(a); X \simeq f(b, Y)$ 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:

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	$c \simeq f(Z, a); X \simeq f(b, Y)$	applying Positive Superposition on 2 and 1
5	$f(Z, a) \not\simeq f(b, Y); c \simeq f(b, Y)$	applying Equality Factoring on 4
6	$c \simeq f(b, a)$	applying Equality Resolution on 5
7	$g(f(b, a)) \not\simeq g(f(X, Y))$	applying Negative Superposition on 6 and 3
8	$\square$	applying 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	$g(a) \simeq f(Z, a)$	applying Instantiation on 2 with $\sigma = \{X \mapsto a\}$
5	$c \simeq f(Z, a); X \simeq f(b, Y)$	applying Positive Replacement on 4 and 1
6	$c \simeq f(Z, a); c \simeq f(b, Y)$	applying Instantiation on 5 with $\sigma = \{X \mapsto c\}$
7	$f(Z, a) \not\simeq f(b, Y); c \simeq f(b, Y)$	applying Identical Equality Factoring on 6
8	$f(b, a) \not\simeq f(b, a); c \simeq f(b, a)$	applying Instantiation on 7 with $\sigma = \{Y \mapsto a; Z \mapsto b\}$
9	$c \simeq f(b, a)$	applying Identical Equality Resolution on 8
10	$g(f(b, a)) \not\simeq g(f(X, Y))$	applying Negative Replacement on 9 and 3
11	$g(f(b, a)) \not\simeq g(f(b, a))$	applying Instantiation on 10 with $\sigma = \{X \mapsto b; Y \mapsto a\}$
12	$\square$	applying 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 λΠ-calculus modulo is therefore

$$\iota : \mathsf{Type}$$
$$o : \mathsf{Type}$$
$$\mathsf{proof} : o \to \mathsf{Type}$$
$$\dot{\simeq} : \iota \to \iota \to o$$
$$\simeq : \iota \to \iota \to \mathsf{Type}$$
$$\simeq \hookrightarrow \lambda x : \iota.\ \lambda y : \iota.\ \Pi p : (\iota \to o).\ \mathsf{proof}\ (p\ x) \to \mathsf{proof}\ (p\ y)$$
$$\dot{\Rightarrow} : o \to o \to o$$
$$\mathsf{proof}\ (\dot{\simeq}\ x\ y) \hookrightarrow \simeq\ x\ y$$
$$\mathsf{proof}\ (\dot{\Rightarrow}\ A\ B) \hookrightarrow \mathsf{proof}\ A \to \mathsf{proof}\ B$$
$$\mathsf{refl} : \Pi x : \iota.\ \simeq\ x\ x$$
$$\mathsf{refl} \hookrightarrow \lambda x : \iota.\ \lambda p : \iota \to o.\ \lambda t : \mathsf{proof}\ (p\ x).\ t$$
$$g : \iota \to \iota$$
$$f : \iota \to \iota \to \iota$$
$$a : \iota$$
$$b : \iota$$
$$c : \iota$$

We first declare the three input clauses:

$$c1 : \Pi X : \iota.\ \Pi Y : \iota.\ \Pi \flat : o.\ (\simeq\ c\ (g\ a) \to \mathsf{proof}\ \flat) \to (\simeq\ X\ (f\ b\ Y) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$

$$c2 : \Pi X : \iota.\ \Pi Z : \iota.\ \Pi \flat : o.\ (\simeq\ (g\ X)\ (f\ Z\ X) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$
$$c3 : \Pi X : \iota.\ \Pi Y : \iota.\ \Pi \flat : o.\ ((\simeq\ (g\ c)\ (g\ (f\ X\ Y)) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$

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

$$c4 : \Pi Z : \iota.\ \Pi \flat : o.\ (\simeq\ (g\ a)\ (f\ Z\ a) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$
$$c4 \hookrightarrow \lambda Z : \iota.\ c2\ a\ Z$$
$$c5 : \Pi X : \iota.\ \Pi Y : \iota.\ \Pi Z : \iota.\ \Pi \flat : o.\ (\simeq\ c\ (f\ Z\ a) \to \mathsf{proof}\ \flat) \to (\simeq\ X\ (f\ b\ Y) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$
$$\begin{array}{l} c5 \hookrightarrow \lambda X : \iota.\ \lambda Y : \iota.\ \lambda Z : \iota.\ \lambda \flat : o.\ \lambda p : (\simeq\ c\ (f\ Z\ a) \to \mathsf{proof}\ \flat).\ \lambda l : (\simeq\ X\ (f\ b\ Y) \to \mathsf{proof}\ \flat). \\ \qquad c1\ X\ Y\ \flat\ (\lambda q :\simeq\ c\ (g\ a).\ c4\ Z\ \flat\ (\lambda r :\simeq\ (g\ a)\ (f\ Z\ a).\ p\ (r\ (\lambda u : \iota.\ \dot{\simeq}\ c\ u)\ q)))\ l \end{array}$$
$$c6 : \Pi Y : \iota.\ \Pi Z : \iota.\ \Pi \flat : o.\ (\simeq\ c\ (f\ Z\ a) \to \mathsf{proof}\ \flat) \to (\simeq\ c\ (f\ b\ Y) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$
$$c6 \hookrightarrow \lambda Y : \iota.\ \lambda Z : \iota.\ c5\ c\ Y\ Z$$
$$c7 : \Pi Y : \iota.\ \Pi Z : \iota.\ \Pi \flat : o.\ ((\simeq\ (f\ Z\ a)\ (f\ b\ Y) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to (\simeq\ c\ (f\ b\ Y) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$
$$\begin{array}{l} c7 \hookrightarrow \lambda Y : \iota.\ \lambda Z : \iota.\ \lambda \flat : o.\ \lambda p : ((\simeq\ (f\ Z\ a)\ (f\ b\ Y) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat).\ \lambda q : (\simeq\ c\ (f\ b\ Y) \to \mathsf{proof}\ \flat). \\ \qquad c6\ Y\ Z\ \flat\ (\lambda r :\simeq\ c\ (f\ Z\ a).\ p\ (r\ (\lambda u : \iota.\ \dot{\Rightarrow}\ (\dot{\simeq}\ u\ (f\ b\ Y))\ \flat)\ q))\ q \end{array}$$
$$c8 : \Pi \flat : o.\ ((\simeq\ (f\ b\ a)\ (f\ b\ a) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to (\simeq\ c\ (f\ b\ a) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$
$$c8 \hookrightarrow c7\ a\ b$$
$$c9 : \Pi \flat : o.\ (\simeq\ c\ (f\ b\ a) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$
$$c9 \hookrightarrow \lambda \flat : o.\ \lambda l : (\simeq\ c\ (f\ b\ a) \to \mathsf{proof}\ \flat).\ c8\ \flat\ (\lambda p : (\simeq (f\ b\ a)\ (f\ b\ a) \to \mathsf{proof}\ \flat).\ p\ (\mathsf{refl}\ (f\ b\ a)))\ l$$
$$c10 : \Pi X : \iota.\ \Pi Y : \iota.\ \Pi \flat : o.\ ((\simeq\ (g\ (f\ b\ a))\ (g\ (f\ X\ Y)) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$
$$\begin{array}{l} c10 \hookrightarrow \lambda X : \iota.\ \lambda Y : \iota.\ \lambda \flat : o.\ \lambda p : ((\simeq\ (g\ (f\ b\ a))\ (g\ (f\ X\ Y)) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat). \\ \qquad c3\ X\ Y\ \flat\ (\ \lambda q : (\simeq\ (g\ c)\ (g\ (f\ X\ Y)) \to \mathsf{proof}\ \flat). \\ \qquad\qquad c9\ \flat\ (\lambda r :\simeq\ c\ (f\ b\ a).\ p\ (r\ (\lambda u.\ \dot{\Rightarrow}\ (\dot{\simeq}\ (g\ u)\ (g\ (f\ X\ Y)))\ \flat)\ q))) \end{array}$$
$$c11 : \Pi \flat : o.\ ((\simeq\ (g\ (f\ b\ a))\ (g\ (f\ b\ a)) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat$$
$$c11 \hookrightarrow c10\ b\ a$$
$$c12 : \Pi \flat : o.\ \mathsf{proof}\ \flat$$
$$c12 \hookrightarrow \lambda \flat : o.\ c11\ \flat\ (\lambda p : (\simeq\ (g\ (f\ b\ a))\ (g\ (f\ b\ a)) \to \mathsf{proof}\ \flat).\ p\ (\mathsf{refl}\ (g\ (f\ b\ a))))$$

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 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 $C_1, \dots, C_m$ with correspond formulas $A_1, \dots A_m$, proving the unsatisfiability of $C_1, \dots, C_m$ amounts to proving the sequent $A_1, \dots, A_m \vdash$ 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 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 `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 λΠ-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 uses also simplification rules that can be used to replace a set of clause by another. For instance, iProver Modulo uses

$$\text{Subsumption Resolution}\ \frac{L; C \qquad \sigma(\overline{L}); \sigma(C); D}{L; C \qquad \sigma(C); D}$$

where $\overline{P} = \neg P$ and $\overline{\neg P} = P$. After the simplification is performed, $\sigma(\overline{L}); \sigma(C); D$ 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

$$\text{Identical Factoring}\ \dfrac{\text{Identical Resolution}\ \dfrac{\text{Instantiation}\ \dfrac{L; C}{\sigma(L); \sigma(C)} \qquad \sigma(\overline{L}); \sigma(C); D}{\sigma(C); \sigma(C); D}}{\sigma(C); D}\ *$$

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 => t` and `x : a -> b`) and can be checked by the dkparse tool. For instance, for the unsatisfiability of the two clauses in Example 1, iProver Modulo outputs:

```
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) =>
                   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))).
clause1 : (bot_var : o -> proof bot_var).
[] clause1 --> (bot_var : o =>
                 clause2 b bot_var (tp : p b a =>
                   clause4 a bot_var (tnp : (p b a -> proof bot_var) => tnp tp))).
```

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

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 translates 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 $nnpp : \Pi p : o.\ \Pi\flat : o.\ ((\mathsf{proof}\ p \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ \flat) \to \mathsf{proof}\ p$. 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 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 C_1 and C_2 leads to C_3 will indeed be a proof that C_1, C_2 and the clausal normal form of $\neg C_3$ is unsatisfiable.

References

[1] Ali Assaf and Guillaume Burel. Translating HOL to Dedukti. Submitted, 2013.

[2] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):1–31, 1994.

[3] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *LNCS*, pages 116–130. Springer, 2011.

[4] Mathieu Boespflug and Guillaume Burel. CoqInE: Translating the calculus of inductive constructions into the λΠ-calculus modulo. In David Pichardie and Tjark Weber, editors, *Second International Workshop on Proof Exchange for Theorem Proving*, 2012.

[5] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus modulo as a universal proof language. In David Pichardie and Tjark Weber, editors, *Second International Workshop on Proof Exchange for Theorem Proving*, 2012.

[6] Guillaume Burel. Experimenting with deduction modulo. In Viorica Sofronie-Stokkermans and Nikolaj Bjørner, editors, *CADE*, volume 6803 of *LNCS*, pages 162–176. Springer, 2011.

[7] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.

[8] Alexis Dorra. Équivalence de Curry-Howard entre le lambda-Pi-calcul et la logique intuitionniste. Rapport de stage de L3, École Polytechnique, 2011.

[9] Gilles Dowek. Polarized resolution modulo. In Cristian S. Calude and Vladimiro Sassone, editors, *IFIP TCS*, volume 323 of *IFIP AICT*, pages 182–196. Springer, 2010.

[10] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*, volume 5 of *Computer Science and Technology Series*. Harper & Row, New York, 1986.

[11] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[12] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[13] Konstantin Korovin. iProver – an instantiation-based theorem prover for first-order logic (system description). In Alessandro Armando and Peter Baumgartner, editors, *IJCAR*, volume 5195 of *LNAI*, pages 292–298. Springer, 2008.

[14] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[15] Alexandre Riazanov and Andrei Voronkov. Vampire 1.1. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning*, volume 2083 of *LNCS*, pages 376–380. Springer, 2001.

[16] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[17] Stephen Schultz. System description: E 0.81. In David A. Basin and Michaël Rusinowitch, editors, *IJCAR*, volume 3097 of *LNCS*, pages 223–228. Springer, 2004.

[18] 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 LNAI, pages 1–12. Springer, 2010.

[19] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.