Understanding Resolution Proofs through Herbrand's Theorem*

Stefan Hetzl[1], Tomer Libal[2], Martin Riener[3], and Mikheil Rukhaia[4]

[1] Institute of Discrete Mathematics and Geometry, Vienna University of Technology
[2] Microsoft Research - Inria Joint Center / École Polytechnique
[3] Institute of Computer Languages, Vienna University of Technology
[4] Inria Saclay / École Polytechnique

Abstract. Computer-generated proofs are usually difficult to grasp for a human reader. In this paper we present an approach to understanding resolution proofs through Herbrand's theorem and the implementation of a tool based on that approach.

The information we take as primitive is which instances have been chosen for which quantifiers, in other words: an expansion tree. After computing an expansion tree from a resolution refutation, the user is presented this information in a graphical user interface that allows flexible folding and unfolding of parts of the proof.

This interface provides a convenient way to focus on the relevant parts of a computer-generated proof. In this paper, we describe the proof-theoretic transformations, the implementation and demonstrate its usefulness on several examples.

1 Introduction

Computer-generated proofs are often difficult to understand for a human reader. This is usually due to a combination of several factors such as the use of deduction formats more suited for proof search than for proof presentation, overwhelming detail in formal proofs, insufficient user interfaces or also extreme proof length. One of the key problems for understanding formal proofs is to distinguish relevant information from irrelevant information.

In systems with quantifiers, such as classical first-order logic, the most important information is typically which instances have been chosen for which quantifiers. On a purely logical level this insight is embodied by Herbrand's theorem [1,2] which characterizes first-order validity in terms of quantifier instances and propositional validity. In this paper we present an approach to understanding computer-generated proofs through the lens of Herbrand's theorem which takes the information about which instances have been chosen for which quantifiers in a proof as fundamental and thus abstracts from the propositional part of the proof.

* Supported by the joint ANR/FWF-project STRUCTURAL, the joint ANR/FWF-project ASAP, the FWF-project ASCOP, the WWTF-project VRG12-04 and the Vienna PhD school in Informatics.

D. Galmiche and D. Larchey-Wendling (Eds.): TABLEAUX 2013, LNCS 8123, pp. 157–171, 2013.

A data structure which is well-suited for representing this information are expansion trees, introduced by Miller in [3]. A first step in distinguishing relevant from irrelevant information is made by displaying an expansion tree instead of a proof in, e.g., a resolution or a tableau calculus. This removes the propositional layer from a proof. However, not all quantifiers are equally important for understanding a proof, for example often we want to consider a proof modulo a simple theory and are hence not interested in the instances of the axioms of that theory. As such distinctions between important and unimportant information depend on the context and are difficult to automate, we let the user decide what information he wants to see in a graphical user interface by allowing a flexible folding and unfolding of expansion trees by point-and-click interactions.

We have implemented our tool in the GAPT-system[1] which is a framework for data structures, algorithms and user interfaces for analyzing and transforming formal proofs. It contains data structures for example formulas, sequents, resolution proofs, sequent proofs and algorithms, e.g., unification, skolemization, cut-elimination, cut-elimination by resolution [4].

The use of Herbrand's theorem for understanding proofs is a well-established technique. It plays the key role in Luckhardt's (manual) analysis [5] of Roth's theorem where it has been used to obtain polynomial bounds (which were obtained independently and by purely mathematical as opposed to logical methods by Bombieri and van der Poorten in [6]). The extraction and analysis of Herbrand-sequents as described by Hetzl et al. [7] has also been used in the computer-assisted analysis of Fürstenberg's topological proof of the infinity of primes by Baaz et al. [8] which yielded Euclid's original argument via cut-elimination. Herbrand's theorem and methods based on it have furthermore also been used in a number of smaller case studies such as [9] by Baaz et al. or [10] by Urban. In the context of the GAPT-system, Herbrand's theorem also plays a central role for the development of techniques for lemma generation, see Hetzl et al. [11,12], and the tool described in this paper is routinely used there.

The general problem of human-readable presentations of computer-generated proofs is well known and a number of other approaches exist in the literature. Horacek [13] presents an approach to transforming computer-generated proofs to a structure that more closely resembles mathematical proofs in natural language. The TRAMP-system by Meier [14] transforms resolution proofs into natural deduction proofs at the assertion level. The interactive derivation viewer IDV by Trac et al. [15] displays a derivation in the TPTP-format as a directed acyclic graph. In [16], Denzinger and Schulz show how to obtain human-readable proof presentations in the context of distributed equational reasoning.

Closest to our approach is the work of Pfenning [17,18], an algorithm for extracting an expansion tree from a resolution refutation by doing grounding, deskolemization and the change of deduction format from refutation to proof in one pass. The contribution of this paper is twofold: we describe a more modular algorithm that first changes the proof format from a resolution refutation to a positive proof in the sequent calculus and only in a second step extracts an

[1] Generic Architecture for Proof Transformations, `http://www.logic.at/gapt`

expansion tree from the sequent calculus proof thus generated. Aside from higher modularity and less implementation effort in the context of the GAPT-system, this procedure has the practical advantage of allowing a translation to a dag-like sequent calculus proof in case grounding of the resolution refutation is too expensive. Secondly we describe the theory and implementation of a convenient graphical user interface for displaying expansion trees which is available on the web.

2 Expansion Trees

The language of first-order logic we consider consists of *variables* and *n-ary function- and predicate- symbols*. As usual, *terms* are built from variables and function symbols in an inductive fashion. 0-ary function symbols are called *constants*. *Formulas* are built from predicates and the connectives $\neg, \wedge, \vee, \rightarrow, \forall, \exists$. A *substitution* is a function mapping variables to terms and its application is extended to terms and formulas in the usual way.

Let A be a formula, an occurrence of a subformula B in A is called *negative* if it is in the scope of an odd number of occurrences of $\neg$ and it is called *positive* otherwise. A quantifier occurrence in a formula A is called *strong* if it is a positive universal or a negative existential and *weak* otherwise. So in other words: the strong quantifiers are exactly those that a transformation to negation normal form would turn into universal quantifiers.

Expansion trees were introduced by Miller in [19,3]. These structures record the substitutions for quantifiers in the original formula and the formulas resulting from instantiations. Informally, an expression $QxA(x) +^{t_1} E_1 +^{t_2} \cdots +^{t_n} E_n$ is an expansion tree, where $Q \in \{\forall, \exists\}$ and $t_1, \ldots, t_n$ are terms such that E_i is again an expansion tree representing $A(t_i)$ for all $i = 1, \ldots, n$.

We deviate from [3] in that we define expansion trees for blocks of quantifiers as this is more natural for display purposes. A vector $(x_1, \ldots, x_k)$ of variables is often abbreviated as $\bar{x}$. Also in contrast to [3] we only consider expansion trees of formulas that do not contain strong quantifiers. In our context of automated deduction, strong quantifiers are removed by Skolemization. As Skolem functions often possess a natural mathematical interpretation (see e.g. [8]), we opt for displaying expansion trees that still include the Skolem functions in order to increase their readability.

Definition 1 (Expansion tree). *Expansion trees and a function* Sh *(for* shallow*) which maps an expansion tree to a formula are defined inductively as follows:*

- *A is an atomic expansion tree for A being an atom and* $\mathrm{Sh}(A) = A$.
- *If E_1, E_2 are expansion trees, then so are $\neg E_1$, $E_1 \wedge E_2$, $E_1 \vee E_2$ and $E_1 \rightarrow E_2$ with* $\mathrm{Sh}(\neg E_1) = \neg \mathrm{Sh}(E_1)$, $\mathrm{Sh}(E_1 \wedge E_2) = \mathrm{Sh}(E_1) \wedge \mathrm{Sh}(E_2)$, *etc.*
- *Let $A(\bar{x})$ be a formula and $\bar{t_1}, \ldots, \bar{t_n}$ ($n \geqslant 1$) be a list of (vectors of) terms. Let $E_1, \ldots, E_n$ be expansion trees with* $\mathrm{Sh}(E_i) = A(\bar{t_i})$ *for $1 \leqslant i \leqslant n$ and let $Q \in \{\forall, \exists\}$, then $Q\bar{x}A(\bar{x}) +^{\bar{t_1}} E_1 +^{\bar{t_2}} \cdots +^{\bar{t_n}} E_n$ is an expansion tree with* $\mathrm{Sh}(Q\bar{x}A(\bar{x}) +^{\bar{t_1}} E_1 +^{\bar{t_2}} \cdots +^{\bar{t_n}} E_n) = Q\bar{x}A(\bar{x})$.

We now define another function Dp (for *deep*) which maps an expansion tree to a quantifier-free formula: its full expansion.

Definition 2. Dp *maps an expansion tree to a formula as follows:*

$$\begin{aligned}
\mathrm{Dp}(E) &= E \text{ for an atomic expansion tree } E,\\
\mathrm{Dp}(\neg E) &= \neg \mathrm{Dp}(E),\\
\mathrm{Dp}(E_1 \circ E_2) &= \mathrm{Dp}(E_1) \circ \mathrm{Dp}(E_2) \text{ for } \circ \in \{\wedge, \vee, \rightarrow\},\\
\mathrm{Dp}(\exists \bar{x} A +^{\bar{t_1}} E_1 +^{\bar{t_2}} \cdots +^{\bar{t_n}} E_n) &= \mathrm{Dp}(E_1) \vee \cdots \vee \mathrm{Dp}(E_n),\\
\mathrm{Dp}(\forall \bar{x} A +^{\bar{t_1}} E_1 +^{\bar{t_2}} \cdots +^{\bar{t_n}} E_n) &= \mathrm{Dp}(E_1) \wedge \cdots \wedge \mathrm{Dp}(E_n).
\end{aligned}$$

In [3], a notion of *expansion proof* was defined from expansion trees using two conditions: *acyclicity* and *tautology*. The acyclicity condition ensures that there are no cycles between the strong quantifier nodes in the expansion tree. Since we deal with formulas that do not contain strong quantifiers, there is no need for this condition.

Definition 3 (Expansion proofs). *An expansion tree E is called an expansion proof of a formula A without strong quantifiers if* $\mathrm{Sh}(E) = A$ *and* $\mathrm{Dp}(E)$ *is a tautology.*

The meaning of expansion proofs is that they encode a proof of validity of the formula they represent. Expansion proofs can be directly translated into sequent calculus or natural deduction proofs, see e.g. [3].

Theorem 1 (Soundness & Completeness). *A formula without strong quantifiers has an expansion proof iff it is valid.*

Proof. In [3]. □

3 Display Expansion Trees

In this section we formally define the data structure of *display expansion tree*, a structure that builds on expansion trees by allowing a flexible *degree* of unfolding. While an expansion tree E induces only the two formulas $\mathrm{Sh}(E)$ and $\mathrm{Dp}(E)$, the display expansion trees based on E turn span the whole spectrum between $\mathrm{Sh}(E)$ and $\mathrm{Dp}(E)$. In our tool, the user will then be able to navigate this spectrum through a comfortable point-and-click interface.

Each quantifier node in a display expansion tree will have one of three states: *open, closed* and *expanded*.

Definition 4 (Display expansion tree). *Display expansion trees are defined as follows:*

- *A is an atomic display expansion tree for an atom A.*
- *If E_1, E_2 are display expansion trees, then so are $\neg E_1$, $E_1 \wedge E_2$, $E_1 \vee E_2$ and $E_1 \rightarrow E_2$.*

– *Let $A(\bar{x})$ be a formula and $\bar{t_1}, \ldots, \bar{t_n}$ ($n \geqslant 1$) be a list of (vectors of) terms. Let $E_1, \ldots, E_n$ be display expansion trees with $\mathrm{Sh}(E_i) = A(\bar{t_i})$ for $1 \leqslant i \leqslant n$ and let $Q \in \{\forall, \exists\}$, then:*
  1. *$Q^{\mathrm{c}}\bar{x}A(\bar{x}) +^{\bar{t_1}} E_1 +^{\bar{t_2}} \cdots +^{\bar{t_n}} E_n$ is a display expansion tree (this block of quantifiers is called* closed*).*
  2. *$Q^{\mathrm{o}}\bar{x}A(\bar{x}) +^{\bar{t_1}} E_1 +^{\bar{t_2}} \cdots +^{\bar{t_n}} E_n$ is a display expansion tree (this block of quantifiers is called* open*).*
  3. *$Q^{\mathrm{e}}\bar{x}A(\bar{x}) +^{\bar{t_1}} E_1 +^{\bar{t_2}} \cdots +^{\bar{t_n}} E_n$ is a display expansion tree (this block of quantifiers is called* expanded*).*

  Under the global side condition: If $Q\bar{x}$ is open or expanded, then all quantifiers between $Q\bar{x}$ and the root must be expanded.

The differences in the status of these quantifier blocks will be apparent once we explain how to show a display expansion tree to a user. To that aim we first define the notion of *display formula*.

Definition 5 (display formula). *Display formulas are defined inductively as follows:*

– *If A is an atom, then A is a display formula.*
– *If A, B are display formulas, then so are $\neg A$, $A \wedge B$, $A \vee B$ and $A \to B$.*
– *If $A(\bar{x})$ is a display formula, then $Q\bar{x}A(\bar{x})$ and $Q\bar{x}\langle \bar{t_1}; \ldots; \bar{t_n}\rangle A(\bar{x})$ are display formulas for $Q \in \{\forall, \exists\}$, where $\bar{t_i}$ are vectors of terms (which represent substitution instances for $\bar{x}$).*
– *If $A_1, \ldots, A_n$ are display formulas, then $\bigwedge\langle A_1, \ldots, A_n\rangle$ and $\bigvee\langle A_1, \ldots, A_n\rangle$ are display formulas for the n-ary connectives $\bigwedge$ and $\bigvee$.*

Given a display expansion tree E what we show to a user is the display formula $\mathrm{Dy}(E)$ defined as follows.

Definition 6. Dy *maps a display expansion tree to a display formula:*

$$\begin{aligned}
\mathrm{Dy}(E) &= E \text{ for atomic } E,\\
\mathrm{Dy}(\neg E) &= \neg\mathrm{Dy}(E),\\
\mathrm{Dy}(E_1 \circ E_2) &= \mathrm{Dy}(E_1) \circ \mathrm{Dy}(E_2) \text{ for } \circ \in \{\wedge, \vee, \to\},\\
\mathrm{Dy}(Q^{\mathrm{c}}\bar{x}A(\bar{x}) +^{\bar{t_1}} E_1 +^{\bar{t_2}} \cdots +^{\bar{t_n}} E_n) &= Q\bar{x}A(\bar{x}) \text{ for } Q \in \{\forall, \exists\},\\
\mathrm{Dy}(Q^{\mathrm{o}}\bar{x}A(\bar{x}) +^{\bar{t_1}} E_1 +^{\bar{t_2}} \cdots +^{\bar{t_n}} E_n) &= Q\bar{x}\langle t_1; \ldots; t_n\rangle A(\bar{x}),\\
\mathrm{Dy}(\exists^{\mathrm{e}}\bar{x}A(\bar{x}) +^{t_1} E_1 +^{\bar{t_2}} \cdots +^{t_n} E_n) &= \bigvee\langle \mathrm{Dy}(E_1), \ldots, \mathrm{Dy}(E_n)\rangle,\\
\mathrm{Dy}(\forall^{\mathrm{e}}\bar{x}A(\bar{x}) +^{t_1} E_1 +^{\bar{t_2}} \cdots +^{t_n} E_n) &= \bigwedge\langle \mathrm{Dy}(E_1), \ldots, \mathrm{Dy}(E_n)\rangle
\end{aligned}$$

The user can hence control the formula that he sees by changing the status of quantifier nodes of a display expansion tree. A display expansion tree in a particular state of partial unfolding lies hence between the shallow formula and the deep formula of the underlying expansion tree. This observation can be made precise as follows:

Definition 7. Fm *maps display formulas to formulas:*

$$\begin{aligned}
\mathrm{Fm}(A) &= A \text{ for a formula } A,\\
\mathrm{Fm}(Q\bar{x}A(\bar{x})) &= Q\bar{x}A(\bar{x}) \text{ for } Q \in \{\forall, \exists\},\\
\mathrm{Fm}(Q\bar{x}\langle t_1; \ldots; t_n\rangle A(\bar{x})) &= Q\bar{x}A(\bar{x}) \text{ for } Q \in \{\forall, \exists\},\\
\mathrm{Fm}(\bigvee\langle A_1, \ldots, A_n\rangle) &= A_1 \vee \cdots \vee A_n,\\
\mathrm{Fm}(\bigwedge\langle A_1, \ldots, A_n\rangle) &= A_1 \wedge \cdots \wedge A_n.
\end{aligned}$$

Proposition 1. *Let E be a display expansion tree. If all quantifiers in E are closed, then* $\mathrm{Fm}(\mathrm{Dy}(E)) = \mathrm{Sh}(E)$. *If all quantifiers in E are expanded, then* $\mathrm{Fm}(\mathrm{Dy}(E)) = \mathrm{Dp}(E)$.

Proof. By induction on the structure of E. □

4 Transforming Resolution Proofs to Expansion Trees

In this section we give an algorithm for constructing expansion proofs from resolution refutations. We proceed by first transforming a resolution refutation of the negation of a formula F into a proof of F in the sequent calculus [20]. The second step is to transform the sequent calculus proof into an expansion tree proof. While the second part is done in a similar way to other sources [3], the first part, to the best of knowledge of the authors, was not described earlier in this form.

The proofs we expect as input to our algorithm are resolution refutations of sets of clauses. A *clause* is a disjunction of literals, a literal is an atom or the negation of an atom. We will sometimes write a clause in the format $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ for B_i being the positive and A_j the negative literals. This notation facilitates the connection to the sequent calculus. We will denote clauses by uppercase Greek letters and formulas by uppercase Latin letters. Substitutions and *most general unifiers* are defined as usual and will be denoted by lowercase Greek letters. Terms are denoted by lowercase Latin letters. We will also write $s[t]$ in order to emphasize that t is a subterm of s.

The version of the *resolution calculus* presented in Fig. 1 forms, if one drops the `(Instance)` rule, the minimal version required in order to obtain completeness for first-order logic with equality. This makes our algorithm applicable, via elementary translations, to the proofs obtained by most resolution theorem provers in the market. The redundant `(Instance)` rule allows us to take advantage of simpler proof formats generated by some theorem provers, such as Prover9. Let C be a set of clauses, a tree over the rules in Fig. 1 with leaves from C is a refutation of C if:

- the root of the tree is the empty clause $\square$.
- when we apply a binary rule on clauses Γ and Δ, their sets of free variables must be disjoint.

$$\frac{A \vee \Gamma \qquad \neg B \vee \Delta}{\Gamma\sigma \vee \Delta\sigma}\ \texttt{(Resolve)}^1 \qquad \frac{A \vee B \vee \Gamma}{A\sigma \vee \Gamma\sigma}\ \texttt{(Factor)}^1$$

$$\frac{t = s \vee \Gamma \qquad A[r] \vee \Delta}{A[s]\sigma \vee \Gamma\sigma \vee \Delta\sigma}\ \texttt{(Paramod)}^2 \qquad \frac{\Gamma}{\Gamma\sigma}\ \texttt{(Variant)}^3$$

$$\frac{\Gamma}{\Gamma\sigma}\texttt{(Instance)} \qquad \frac{}{x = x}\texttt{(Reflexivity)}$$

1. σ is a most general unifier of A and B.
2. σ is a most general unifier of t and r.
3. σ is a variable renaming.

Fig. 1. The resolution calculus

4.1 Transforming Resolution Proofs to Sequent Proofs

The calculus we will use is presented in Fig. 2. It extends the classical sequent calculus [20] with additional equality rules. The rules for the connectives $\exists, \vee$ and $\rightarrow$ are analogous to the ones presented. We will denote multisets of formulas by uppercase Latin letters. *sequents* are pairs of multisets of formulas denoted by $\Gamma \vdash \Delta$. The formulas in the upper sequents that do not occur in the lower sequent are called *auxiliary formulas* of the rule, those in the lower sequent are called the *principal formulas*.

In order to be able to relate refutations with proofs, we require the following auxiliary definitions.

$$\frac{}{A \vdash A}\ \texttt{(Ax)} \qquad \frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg A}\ (\neg : r) \qquad \frac{\Gamma \vdash \Delta, A}{\neg A, \Gamma \vdash \Delta}\ (\neg : l)$$

$$\frac{\Gamma \vdash \Delta, A \qquad \Lambda \vdash \Pi, B}{\Gamma, \Lambda \vdash \Delta, \Pi, A \wedge B}\ (\wedge : r) \qquad \frac{A, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta}\ (\wedge : l_1) \qquad \frac{B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta}\ (\wedge : l_2)$$

$$\frac{\Gamma \vdash \Delta, A[x]}{\Gamma \vdash \Delta, \forall y.A[y]}\ (\forall : r)^1 \qquad \frac{A[t], \Gamma \vdash \Delta}{\forall y.A[y], \Gamma \vdash \Delta}\ (\forall : l)^2 \qquad \frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}\ (Contr : r)$$

$$\frac{A, A, \Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}\ (Contr : l) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}\ (Weak : r) \qquad \frac{\Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}\ (Weak : l)$$

$$\frac{\Gamma \vdash \Delta, A \qquad A, \Lambda \vdash \Pi}{\Gamma, \Lambda \vdash \Delta, \Pi}\ (Cut) \qquad \frac{}{\vdash t = t}\ \texttt{(Reflexivity)}$$

$$\frac{\Gamma \vdash \Delta, t = s \qquad \Lambda \vdash \Pi, A[s]}{\Gamma, \Lambda \vdash \Delta, \Pi, A[t]}\ (Eq : r) \qquad \frac{\Gamma \vdash \Delta, t = s \qquad A[s], \Lambda \vdash \Pi}{A[t], \Gamma, \Lambda \vdash \Delta, \Pi}\ (Eq : l)$$

1. x does not occur free in Γ, Δ or in $\forall y.A[y]$.
2. t does not contain variables bound in A.

Fig. 2. The sequent calculus

Definition 8. *Given two sequents s_1 and s_2 of the forms $\Gamma \vdash \Delta$ and $\Pi \vdash \Lambda$, their product $s_1 \times s_2$ is $\Gamma, \Pi \vdash \Delta, \Lambda$. Given two sets of sequents S_1 and S_2, their product $S_1 \times S_2$ is the set containing all possible products between sequents of each set.*

Definition 9 (Clause normal forms). *Let A, A_1, A_2 denote formulas without weak quantifiers and B, B_1, B_2 denote formulas without strong quantifiers and let P denote atoms. Define the mappings $\mathit{CNF}^+(A)$ and $\mathit{CNF}^-(B)$ by the following mutual induction:*

$$\begin{array}{ll}
\mathit{CNF}^+(P) = \{\vdash P\} & \mathit{CNF}^+(A_1 \wedge A_2) = \mathit{CNF}^+(A_1) \cup \mathit{CNF}^+(A_2) \\
\mathit{CNF}^-(P) = \{P \vdash\} & \mathit{CNF}^+(A_1 \vee A_2) = \mathit{CNF}^+(A_1) \times \mathit{CNF}^+(A_2) \\
\mathit{CNF}^+(\neg B) = \mathit{CNF}^-(B) & \mathit{CNF}^-(B_1 \wedge B_2) = \mathit{CNF}^-(B_1) \times \mathit{CNF}^-(B_2) \\
\mathit{CNF}^-(\neg A) = \mathit{CNF}^+(A) & \mathit{CNF}^-(B_1 \vee B_2) = \mathit{CNF}^-(B_1) \cup \mathit{CNF}^-(B_2) \\
\mathit{CNF}^+(\forall x.A) = \mathit{CNF}^+(A) & \mathit{CNF}^-(\exists x.B) = \mathit{CNF}^-(B)
\end{array}$$

The case of $\rightarrow$ is defined by combining the cases of $\vee$ and $\neg$.

The point of the above two transformation is that $\mathtt{CNF}^+(A)$ is logically equivalent to A while $\mathtt{CNF}^-(B)$ is logically equivalent to $\neg B$, this definition hence avoids an explicit transformation to negation-normal form before computing a clause set. This transformation is extended to sequents as follows:

Definition 10 (Clause normal forms of sequents). *Let $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ be a sequent without strong quantifiers, then $\mathit{CNF}^-(A_1, \ldots, A_n \vdash B_1, \ldots, B_m) = \mathit{CNF}^+(A_1 \wedge \ldots \wedge A_n \wedge \neg B_1 \wedge \ldots \wedge \neg B_m)$.*

While this transformation is exponential in the worst case it has not created problems for our practical applications as the sequents we consider are typically quite close to a conjunctive normal form. In case it does pose a performance-problem this transformation can easily be replaced by the polynomial structural clause form transformation.

Example 1. Let $\Gamma \vdash \Delta = P(a), \forall x.(P(x) \rightarrow Q(x)) \vdash Q(a)$, then

$$\mathtt{CNF}^-(\Gamma \vdash \Delta) = \{\vdash P(a);\ P(x) \vdash Q(x);\ Q(a) \vdash\}$$

The following algorithm generates a sequent proof of $A, \Pi \vdash \Lambda$ when the clause $\Pi \vdash \Lambda$ is in $\mathtt{CNF}^+(A)$.

Data: Formula A and a clause $\Pi \vdash \Lambda$ such that $\Pi \vdash \Lambda \in \mathtt{CNF}^+(A)$
begin
  A *is an atom* $\implies A \vdash A$
  $A = \neg B \implies$ apply $(\neg : l)$ to $\mathtt{PCNF}^-(\Pi \vdash \Lambda, B)$
  $A = B \wedge C \implies$ apply either $(\wedge : l_1)$ to $\mathtt{PCNF}^+(B, \Pi \vdash \Lambda)$ or $(\wedge : l_2)$ to $\mathtt{PCNF}^+(C, \Pi \vdash \Lambda)$
  $A = \forall x.A \implies$ apply $(\forall : l)$ to $\mathtt{PCNF}^+(A, \Pi \vdash \Lambda)$
  ...
end

Algorithm 1. $\mathtt{PCNF}^+(A, \Pi \vdash \Lambda)$

The dual algorithm $\mathtt{PCNF}^-$ for computing a sequent calculus proof in the case of $\Pi \vdash \Lambda \in \mathtt{CNF}^-(A)$ is defined in a similar way and we obtain:

Lemma 1. *Let A be a formula without weak quantifiers and B be a formula without strong quantifiers and $\Pi \vdash \Lambda$ a clause, then:*

- *if $\Pi \vdash \Lambda \in \mathtt{CNF}^+(A)$ then $\mathtt{PCNF}^+(A, \Pi \vdash \Lambda)$ is a sequent proof of $A, \Pi \vdash \Lambda$.*
- *if $\Pi \vdash \Lambda \in \mathtt{CNF}^-(B)$ then $\mathtt{PCNF}^-(\Pi \vdash \Lambda, B)$ is a sequent proof of $\Pi \vdash \Lambda, B$.*

Proof. By a straightforward induction on the structure of A. □

For actual applications it is more useful to generate proofs of sequents, not just of formulas, i.e. we work in a setting where a sequent $\Gamma \vdash \Delta$ takes the role of the formulas A or B above. To that aim the above algorithms are extended in a straightforward way to an algorithm $\mathtt{PCNF}(\Gamma \vdash \Delta, \Pi \vdash \Lambda)$ that generates a sequent calculus proof of $\Gamma, \Pi \vdash \Delta, \Lambda$ if $\Pi \vdash \Lambda \in \mathtt{CNF}^-(\Gamma \vdash \Delta)$.

Lemma 2. *Let $\Gamma \vdash \Delta$ be a sequent without strong quantifiers and $\Pi \vdash \Lambda \in \mathtt{CNF}^-(\Gamma \vdash \Delta)$, then $\mathtt{PCNF}(\Gamma \vdash \Delta, \Pi \vdash \Lambda)$ is a proof of $\Gamma, \Pi \vdash \Delta, \Lambda$.*

Proof. If $\Pi \vdash \Lambda \in \mathtt{CNF}^-(\Gamma \vdash \Delta)$, then either $\Pi \vdash \Lambda \in \mathtt{CNF}^+(A)$ for some $A \in \Gamma$ or $\Pi \vdash \Lambda \in \mathtt{CNF}^-(B)$ for some $B \in \Delta$ and we can use Lemma 1. □

Example 2. Continuing Example 1 we have $\mathtt{PCNF}(\Gamma \vdash \Delta, \vdash P(a)) =$

$$\dfrac{P(a) \vdash P(a)}{P(a), \forall x.(P(x) \to Q(x)) \vdash Q(a), P(a)}\ (Weak : *)$$

and $\mathtt{PCNF}(\Gamma \vdash \Delta, P(x) \vdash Q(x)) =$

$$\dfrac{\dfrac{\dfrac{P(x) \vdash P(x) \quad Q(x) \vdash Q(x)}{P(x), P(x) \to Q(x) \vdash Q(x)}\ (\to : l)}{P(x), \forall x.(P(x) \to Q(x)) \vdash Q(x)}\ (\forall : l)}{P(x), P(a), \forall x.(P(x) \to Q(x)) \vdash Q(a), Q(x)}\ (Weak : *)$$

and $\mathtt{PCNF}(\Gamma \vdash \Delta, Q(a) \vdash) =$

$$\dfrac{Q(a) \vdash Q(a)}{Q(a), P(a), \forall x.(P(x) \to Q(x)) \vdash Q(a)}\ (Weak : *)$$

The last algorithm in this section combines the sequent calculus proofs obtained by PCNF and a refutation of $\mathtt{CNF}^-(\Gamma \vdash \Delta)$ into a sequent calculus proof of $\Gamma \vdash \Delta$. This algorithm translates a dag-like refutation into a tree-like proof and hence grounds it. The most important step is to replace a resolution inference by an atomic cut on instances of the sequent calculus proofs obtained from the premises of the resolution inference.

Data: a refutation R of $\mathtt{CNF}^-(\Gamma \vdash \Delta)$
R match **begin**
  An initial clause $\Pi \vdash \Lambda$ $\implies \mathtt{PCNF}(\Gamma \vdash \Delta, \Pi \vdash \Lambda)$
  R is obtained by (`Factor`) with m.g.u. σ from R' $\implies$ apply $(Contr : r)$ or $(Contr : l)$ to $\mathtt{LK}(R')\sigma$
  R is obtained by (`Resolve`) with m.g.u. σ from R_1 and R_2 $\implies$ apply (Cut) to $\mathtt{LK}(R_1)\sigma$ and $\mathtt{LK}(R_2)\sigma$
  ...
end

Algorithm 2. LK

Theorem 2. *Let $\Gamma \vdash \Delta$ be a sequent without strong quantifiers and let R be a refutation of $\mathtt{CNF}^{-}(\Gamma \vdash \Delta)$, then $\mathtt{LK}(R)$ is a sequent calculus proof of $\Gamma \vdash \Delta$.*

Proof. By a straightforward induction on the structure of the refutation. □

4.2 Transforming Sequent Proofs to Expansion Trees

In this section we describe how to read off expansion trees from sequent calculus proofs. The algorithm presented in this section is based on [3] but in addition deals with quantifier-free cuts and equation rules. The algorithm `merge` used in this algorithm for the merging of two expansion trees is defined in [3].

Data: A sequent proof P
if (Ax) **then**
  return an atomic expansion tree for each formula
else
  return expansion trees of upper sequents and replace the trees E_i of the auxiliary formulas with P match **begin**
    $(\wedge : r) \Longrightarrow E_1 \wedge E_2$
    $(\forall : l)$ *with principal formula* $\forall x.A$ *and auxiliary formula* $A[t/x] \Longrightarrow \forall x.A +^t E$
    $(Contr : l) \Longrightarrow \mathtt{merge}(E_1, E_2)$
    $(Cut) \Longrightarrow \varnothing$
    $(Eq : r) \Longrightarrow E_2$
    ...
  end
end

Algorithm 3. ET

Theorem 3. *Let P be a sequent proof of s without strong quantifiers, then $\mathtt{ET}(P)$ is a sequent of expansion trees of the formulas in s.*

Proof. By a straightforward induction on the structure of P. □

4.3 Complexity and Scalability

Expansion trees are an inherently ground formalism. This has the consequence that the translation from a (non-ground) resolution refutation to an expansion tree is expontential in the worst case. On the one hand this is a limitation of the algorithms presented here. On the other hand, grounding has a signifcant benefit: the witnesses which typically carry important information can only be read off from a ground proof. We will illustrate this phenomenon in the next section by describing an example for a clause set whose refutation only shows that a certain puzzle can be solved while its expansion tree contains the solution (which necessarily must be a ground term).

An extension of our algorithms which would be useful for such critical cases would be to carry out the computation of ground instances *on demand*. The present user-interface would not change but the implementation of a display expansion tree would: instead of keeping a complete expansion tree in memory, it would only store the original resolution refutation and compute the ground instances of single quantifiers when asked to.

5 Implementation and Examples

Programmed in Scala, GAPT is a framework intended, on the one hand, to allow an easy and intuitive programming of proof theoretical algorithms and applications and on the other hand to be as general and flexible as possible, in order to be able to target the widest range of languages and calculi. To meet these two requirements, we make extensive use of Scala's object-oriented (OO) and functional paradigms. The OO support is used mainly in order to build a complex type system and abstraction between different logics. The functional support is used, as we will see in the rest of this paper, in order to map formulas, proofs and similar data directly to Scala functions and algebraic data structures. The fact that Scala is compatible with Java allowed us to use its built-in libraries in order to supply a comprehensive graphical user interface. GAPT supplies algebraic data structures for terms, formulas, sequents, resolution proofs, sequent proofs and many other logical objects. The functionality described in this paper is available from version 1.4 on. The interested reader is invited to download the current version from `http://www.logic.at/gapt`.

5.1 Import of a Resolution Proof

The GAPT-System contains two methods for importing resolution proofs. The first is based on proof replaying by reproving each inference through forward reasoning [21] in the minimal resolution calculus implemented in GAPT (Fig. 1). A drawback of this method is that the resulting proofs may differ significantly from the ones found by the theorem prover and that search might be inefficient for macro-rules like hyperresolution. Therefore the second method implements a direct import from the format for the Ivy proof checker [22]. Ivy's resolution calculus (Fig. 3) replaces unification by an explicit instance rule which applies the substitution separately. The next step in the extraction of an expansion tree - the transformation to LK - grounds the proof, applying the substitution to the respective subproofs anyway. Therefore we decided to add the instance rule to GAPT's resolution calculus instead of merging instance rules into unifiers within the other inference rules. The flip rule is expanded to a proof of equational symmetry from the equational reflexivity axiom.

The conversion of Prover9's output [23] to the Ivy format is performed by prooftrans which is part of Prover9's LADR distribution. An Ivy proof is represented as a Lisp S-Expression which requires a different naming convention, therefore GAPT's parser has to integrate proper renaming of constants and variables according to these conventions.

We will use a running example to illustrate the work-flow of our tool. The running example is the famous puzzle of a farmer who wants to transport a wolf, a goat and a cabbage across a river using a boat in which he can take at most one of these items with him. The difficulty is that he can neither leave goat and cabbage nor wolf and goat alone on one of the shores. How can he cross the river? This is formalized as problem `PUZ047+1` of the TPTP-library [24]. The formalization uses a 5-ary predicate symbol p whose first four coordinates contain the

$$\frac{A \vee \Gamma \qquad \neg A \vee \Delta}{\Gamma \vee \Delta}\,\texttt{(Resolve)} \qquad \frac{A \vee A \vee \Gamma}{A \vee \Gamma}\,\texttt{(Factor)}$$

$$\frac{t = s \vee \Gamma \qquad A[t] \vee \Delta}{A[s] \vee \Gamma \vee \Delta}\,\texttt{(Paramod)} \qquad \frac{\Gamma}{\Gamma\sigma}\,\texttt{(Instance)}^{1}$$

$$\frac{\Gamma, s = t}{\Gamma, t = s}\,\texttt{(Flip)} \qquad \frac{}{x = x}\,\texttt{(Reflexivity)}$$

1. σ is an arbitrary substitution.

Fig. 3. The Ivy Resolution calculus

positions of farmer, wolf, goat and cabbage and whose fifth position contains the actions already taken. Universally quantified implications describe the possible actions, the additional axiom $\mathrm{p}(\mathrm{south}, \mathrm{south}, \mathrm{south}, \mathrm{south}, \mathrm{start})$ describes the initial state, the goal is to prove $\exists z\, \mathrm{p}(\mathrm{north}, \mathrm{north}, \mathrm{north}, \mathrm{north}, z)$. For testing our running example we first produce a Prover9 output file by running:

```
$ tptp_to_ladr < PUZ047+1.p > PUZ047+1.in
$ prover9 < PUZ047+1.in > PUZ047+1.out
```

Then we start the command-line interface of GAPT and load the resolution proof from the output file.

```
$ ./cli.sh
scala> val p = loadProver9Proof( "PUZ047+1.out" )
```

This command imports a resolution refutation `p`. One important aspect of this refutation is that **it does not contain the solution to the puzzle**, it merely shows that the puzzle is solvable. The actual solution will be computed by our tool automatically by the transformation of the resolution refutation to an expansion tree. The solution will then be presented in plain sight to the user as instance of the above-mentioned existential quantifier. This example thus illustrates very well the added value of expansion trees over resolution refutations.

5.2 Extraction of an Expansion Tree

As described in Section 4, the extraction of an expansion tree from a resolution refutation proceeds in two phases. First we import a resolution refutation and transform it into a sequent calculus proof following Algorithms 1 and 2.

In addition to the resolution proof, the original input formula is extracted from Prover9's output file. The rationale behind this is that the user expects to see an expansion tree representing the input formula, its clause normal form might be of a significantly different shape. The original formula is transformed into a sequent which forms the end-sequent of the sequent calculus proof that is constructed. This can be carried out by

```
scala> val q = loadProver9LKProof( "PUZ047+1.out" )
```

which creates a sequent calculus proof `q` from the refutation in `PUZ047+1.out`. The expansion trees can then be read off from this sequent calculus proof by:

```
scala> val E = extractExpansionTrees( q )
```

5.3 The Graphical User Interface

ProofTool is the *Graphical User Interface* of the GAPT system [25]. It can be used in two ways: as a pure visualization tool (with the features like zooming, scrolling, searching, etc.) and as a proof manipulator (allowing to call GAPT's proof transformations such as cut-elimination, regularization, skolemization, etc.). The objects ProofTool can render are formulas, sequents, proofs, trees, sequent- and definition-lists. Sequents consisting of expansion trees are handled in a special way to support the interactive visualization features specific to expansion trees. A sequent of expansion trees is displayed in a two column split pane, where one column is for the antecedent and the other is for the consequent of the sequent.

Expansion trees are displayed in ProofTool in the following way: For each expansion tree a display expansion tree is produced by adding the state "closed" to the quantifier nodes occurring in the expansion tree. Then the display formula of the display expansion tree is rendered on the screen. Finally, the display formula can be manipulated by changing the state of quantifiers. A single left-click changes the state from closed to open, from open to expanded, and from expanded to closed. Additionally a context-menu is opened on a right-click to allow a direct state-change.

The expansion trees of our running example can be displayed by

```
scala> prooftool( E )
```

which allows the solution to the puzzle to be read off with a single click:

$$\text{take\_goat}(\text{go\_alone}(\text{take\_wolf}(\text{take\_goat}(\text{take\_cabbage}(\text{go\_alone}(\text{take\_goat}(\text{start})))))))$$

In order to illustrate the display of an expansion tree with nested quantifiers we include a screen-shot of a simple example in Figure 4.

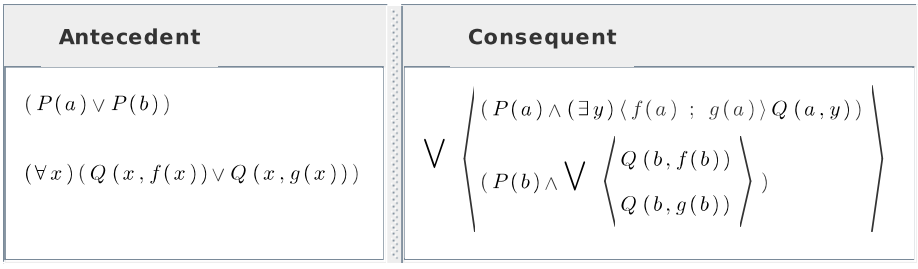

Fig. 4. An expansion of the sequent $P(a) \vee P(b), \forall x(Q(x, f(x)) \vee Q(x, g(x))) \vdash \exists x(P(x) \wedge \exists y Q(x, y))$

6 Conclusion

We have described an approach to understanding resolution proofs through Herbrand's theorem and a tool based on this approach. We have illustrated its usefulness on two examples. The computation of ground instances of quantifiers in combination with a flexible display of an expansion tree in a graphical user interface allows a very quick access to the crucial turning points of a computer-generated proof.

There are several important lines for future work: definitions (i.e. abbreviations of formulas by new predicate symbols) are crucial for human-readable formalizations of mathematical proofs. They can be integrated into this approach in a straightforward way by allowing to fold and unfold them too. Furthermore, just as the original notions of expansion trees [3], our approach and implementation supports higher-order logic as well. The only part of the programs not supporting higher-order logic is the transformation from refutations to sequent proofs, which is customized to the first-order resolution calculus. Another highly interesting extension is to carry out the computation of ground instances on demand as described in Section 4.3. This can be continued much beyond the scope of resolution proofs. For example, by relying on the relationship between cut-elimination and tree grammars established in [26] it would even be possible to do cut-elimination on demand by computing only the instances of a certain quantifier from a proof with cuts.

References

1. Herbrand, J.: Recherches sur la théorie de la démonstration. PhD thesis, Université de Paris (1930)
2. Buss, S.R.: On Herbrand's Theorem. In: Leivant, D. (ed.) LCC 1994. LNCS, vol. 960, pp. 195–209. Springer, Heidelberg (1995)
3. Miller, D.: A Compact Representation of Proofs. Studia Logica 46(4), 347–370 (1987)
4. Baaz, M., Leitsch, A.: Cut-elimination and Redundancy-elimination by Resolution. Journal of Symbolic Computation 29(2), 149–176 (2000)
5. Luckhardt, H.: Herbrand-Analysen zweier Beweise des Satzes von Roth: Polynomiale Anzahlschranken. Journal of Symbolic Logic 54(1), 234–263 (1989)
6. Bombieri, E., van der Poorten, A.: Some quantitative results related to Roth's theorem. Journal of the Australian Mathematical Society 45(2), 233–248 (1988)
7. Hetzl, S., Leitsch, A., Weller, D., Woltzenlogel Paleo, B.: Herbrand Sequent Extraction. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC/Calculemus/MKM 2008. LNCS (LNAI), vol. 5144, pp. 462–477. Springer, Heidelberg (2008)
8. Baaz, M., Hetzl, S., Leitsch, A., Richter, C., Spohr, H.: CERES: An Analysis of Fürstenberg's Proof of the Infinity of Primes. Theoretical Computer Science 403(2-3), 160–175 (2008)
9. Baaz, M., Hetzl, S., Leitsch, A., Richter, C., Spohr, H.: Cut-Elimination: Experiments with CERES. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 481–495. Springer, Heidelberg (2005)

10. Urban, C.: Classical Logic and Computation. PhD thesis, University of Cambridge (October 2000)
11. Hetzl, S., Leitsch, A., Weller, D.: Towards Algorithmic Cut-Introduction. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18. LNCS, vol. 7180, pp. 228–242. Springer, Heidelberg (2012)
12. Hetzl, S.: Project Presentation: Algorithmic Structuring and Compression of Proofs (ASCOP). In: Jeuring, J., Campbell, J.A., Carette, J., Dos Reis, G., Sojka, P., Wenzel, M., Sorge, V. (eds.) CICM 2012. LNCS, vol. 7362, pp. 438–442. Springer, Heidelberg (2012)
13. Horacek, H.: Presenting Proofs in a Human-Oriented Way. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 142–156. Springer, Heidelberg (1999)
14. Meier, A.: System Description: TRAMP: Transformation of Machine-Found Proofs into ND-Proofs at the Assertion Level. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 460–464. Springer, Heidelberg (2000)
15. Trac, S., Puzis, Y., Sutcliffe, G.: An interactive derivation viewer. Electronic Notes in Theoretical Computer Science 174(2), 109–123 (2007)
16. Denzinger, J., Schulz, S.: Recording, Analyzing and Presenting Distributed Deduction Processes. In: Hong, H. (ed.) 1st International Symposium on Parallel Symbolic Computation (PASCO). Lecture Notes Series in Computing, vol. 5, pp. 114–123. World Scientific Publishing (1994)
17. Pfenning, F.: Analytic and non-analytic proofs. In: Shostak, R.E. (ed.) CADE 1984. LNCS, vol. 170, pp. 394–413. Springer, Heidelberg (1984)
18. Pfenning, F.: Proof Transformations in Higher-Order Logic. PhD thesis, Carnegie Mellon University (1987)
19. Miller, D.: Proofs in Higher-Order Logic. PhD thesis, Carnegie-Mellon University (1983)
20. Gentzen, G.: Untersuchungen über das logische Schließen I. Mathematische Zeitschrift 39(2), 176–210 (1934)
21. Dunchev, C., Leitsch, A., Libal, T., Riener, M., Rukhaia, M., Weller, D., Woltzenlogel-Paleo, B.: System Feature Description: Importing Refutations into the GAPT Framework. In: Proof Exchange for Theorem Proving Second International Workshop, PxTP (2012)
22. Mccune, W., Shumsky, O.: Ivy: A Preprocessor And Proof Checker For First-Order Logic. In: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000)
23. McCune, W.: Prover9 and mace4 manual - output files (2005-2010), `https://www.cs.unm.edu/~mccune/mace4/manual/2009-11A/output.html`
24. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)
25. Dunchev, C., Leitsch, A., Libal, T., Riener, M., Rukhaia, M., Weller, D., Woltzenlogel-Paleo, B.: ProofTool: GUI for the GAPT Framework (to appear)
26. Hetzl, S.: Applying tree languages in proof theory. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 301–312. Springer, Heidelberg (2012)