Proof Generalization and Function Introduction
(extended abstract)

Nicolas Peltier
LEIBNIZ-IMAG
46, avenue Félix Viallet, 38031 Grenoble Cedex, FRANCE
Nicolas.Peltier@imag.fr

1 Introduction and Overview of the Paper

In this work, we extend a former method for generalizing proofs in first-order logic and we show how to combine it with some lemma generation techniques based on function introduction [1]. The approach is presented in the context of the resolution method, but its basic principles can be applied to others calculi, such as tableaux, the connection method, or the sequent calculus. The main application of this proof generalization method is analogical reasoning (see [5]): Proofs are first transformed into more general ones, then stored into a database containing theorems and proofs. When a new conjecture is presented, the system will look for more a general formula into the database, using a specialized matching algorithm [5]. The generalization step is one of the most important steps of analogy detection. It aims at extracting the relevant information from the proof, in order to increase the number of conjectures that can be matched and to make the matching step as easy as possible.

The first sections are devoted to a brief description of the proof generalization method. First, we introduce a generalization ordering between unsatisfiable sets of clauses. The order relation is the same as in our previous work [4, 5]. Then, we give an algorithm for transforming formulae into more general ones. The proposed algorithm is based on similar ideas to, but strictly more powerful than, the original one presented in [4].

Then we combine the proposed method with the so-called function introduction rule described in [1]. As proven in [1], this rule may decrease significantly the length of a proof and improves its readability. It can be applied either during proof search, or as a post-processing step, on an already obtained proof [9]. We show how to extend the generalization algorithm in order to take into account the use of the function introduction rule.

These results have important practical consequences for analogical reasoning: first, we obtain a new generalization algorithm, strictly more powerful than the original one. Secondly, shorter and more structured generalized proofs can be obtained, thanks to the use of the function introduction rules [1, 9].

2 The generalization ordering

Generalizing proofs means transforming the proof of a theorem $A$ into a new proof of a more general theorem $B$. Therefore, before presenting our algorithm for generalizing
proves, we first have to precisely define what we mean by “more general formulae”. What do we mean by saying that a formula $A$ is more general than another formula $B$? A first, obvious attempt could be to use a purely semantic approach, i.e. to replace “more general” by “logically implies” but this would make this relation undecidable, hence far to general for being usable in practice. Moreover, we contend that the logical values of the formulae $A, B$ are not sufficient criteria for defining this notion of generality. Indeed, it is clear that the proofs of $A$ and $B$ must be related in some way, independently of their semantic.

For example, let us consider the formulae $A \equiv \forall k \sum_{i=0}^{k-1} i \times i + 1 = k^2$ and $B \equiv \forall k \exists x \sum_{i=0}^{k-1} i \times i + 1 = x^2$. From a logical point of view, $A$ and $B$ are equivalent (in Peano’s arithmetic). However, it is clear, from an intuitive point of view, that the former formula is more general than the latter. Indeed, $B$ simply says that for all $k$, there exists an $x$ such that the sum of the $k$ first odd integers is equal of $x^2$, whereas $A$ gives us the value of $x$ (which is equal to $k$). Any proof of $A$ can be effectively transformed into a proof of $B$ (the converse being not true). Therefore $A$ gives us more precise information, or simply more information than $B$. Otherwise stated, for us “more general” means “giving more information” which corresponds to the standard mathematical use.

Similarly, we do not say that a logical formula $F_1$ is more general than $F_2$ if the proofs of $F_1$ and $F_2$ are not related in some way. For example, $F_1 : \forall y. B(y) \lor \top$ will not be said to be more general than $F_2 : \exists x. A(x) \lor \forall y. \neg A(y)$ despite the fact that $F_1 \models F_2$, since the (cut-free) proof of $F_2$ (which is a valid formula) is not related to the one of $F_1$. Indeed, the proofs of these two formulae have, from an intuitive point of view “nothing in common”.

Hence, we see the notion of generality, although included in the one of the logical consequence, as more restrictive. A purely semantic approach is therefore not sufficient. We have to use a syntactical approach. Intuitively, we will say that $A$ is more general than $B$ iff any proof of $A$ can be automatically transformed (in some efficient way) into a proof $B$. In order to define it on a formal basis, we introduce the following definitions.

**Definition 1** A set of clauses $S$ $H$-subsumes a set of clauses $S'$ iff each ground instances of a clause in $S'$ is subsumed by a clause in $S$.

**Definition 2** An (unsatisfiable) set of clauses $S$ will be said to be more general than another set of clauses $S'$ iff $S'$ $H$-subsumes $S$. This relation will be denoted by $S >_{gen} S'$ in the following.

At first glance, it may seem strange that we obtain a “more general” set of clauses by taking an instance of one of the clauses. However, one should remember that resolution is a refutation procedure, thus considering the negation of the formula to prove, (more precisely, the negation of the conclusion) instead of the formula itself. Therefore, instantiating a clause corresponds in some sense to replace universal quantifiers by existential quantifiers in the negation of the formula, hence to replace existential quantifiers by universal quantifiers, which is more likely to be closer to the usual intuitive meaning of “generalisation”. For instance the above set of clauses $S$ corresponds to the formula (before negation and skolemization) $\forall y. (P(y) \lor \exists x. \neg P(x))$, whereas the obtained set of clauses after generalization corresponds to $\forall y. (\neg P(y) \lor P(y))$. The latter is obviously more general than the former since an existential quantified variable has been instantiated: $\forall y. (P(y) \lor \exists x. \neg P(x))$ only says that there exists an $x$ such that $P(x)$, whereas the the formula $\forall y. (\neg P(y) \lor P(y))$ gives explicitly the value of the variable $x$ (here $x = y$).
Example 1 Let $A \equiv \{ A(b), \neg A(b) \lor B, \neg B \lor \neg C, C \}$ and $B \equiv \{ \forall x. A(x), \forall x. \neg A(x) \lor B, \neg B, C \}$.

We have $A >_{gen} B$.

Proposition 1 $>_{gen}$ is a pre-order.

Proposition 2 If $A >_{gen} B$, then $B \models A$. Moreover, any refutation of $A$ can be automatically transformed (in polynomial time) into a refutation of $B$.

3 The generalization algorithm

The principle of the generalization algorithm is to use the information deduced from the resolution proof of $S$ in order to transform $S$ into a more general set of clauses. The idea is to weaken the conditions imposed on the clauses in $S$, such that the “soundness” of the derivation leading to an empty clause will be preserved. Before describing it on a formal basis, we briefly explain the principle of the generalization process on some very simple examples.

Example 2 Let $S$ be the set of clauses $S = \{ \forall x. P(x), \neg P(a) \}$. The resolution proof is obvious: $\forall x. P(x), \neg P(a) \rightarrow \Box$ (with $x \rightarrow a$). Here it is clear that the clause $\forall x. P(x)$ is not needed for deriving a contradiction. What is needed here is only one particular instance of this clause, namely the clause $P(a)$. Hence $S$ can be transformed into: $S' = \{ P(a), \neg P(a) \}$, which is more general than $S$.

Example 3 Let us consider once more the formulae $A$ and $B$ introduced in the Introduction. After negation and skolemization, $A$ and $B$ can be respectively transformed into: $A' \equiv \sum_{i=0}^{k-1} 2 \times i + 1 \neq k^2$ and $B' \equiv \forall x. \sum_{i=0}^{k-1} 2 \times i + 1 \neq x^2$ (where $k$ is a constant symbol). If we consider the refutation of $B'$ (in Peano’s arithmetic) we see that the variable $x$ is only instantiated with one value: $x \rightarrow k$. Hence, we replace $x$ by $k$ in $B'$, thus giving the formula $A'$.

This example shows how the our formal definition of generality relates to the intuitive description given in Introduction. Here the formula $B$ can be automatically transformed into the more general formula $A$, using the information deduced from its proof.

The main application of this notion of generality is the reuse of proofs. By Proposition 2, if $S$ and $S'$ are two sets of clauses such that $S$ is unsatisfiable and $S >_{gen} S'$, then $S'$ is unsatisfiable and any refutation of $S$ can be efficiently transformed into a refutation of $S'$ (it suffices to remove irrelevant subpart of the proof and then to use lifting).

Checking whether a given set of clauses $S$ is more general than another clauses set $S'$ only requires a $H$-subsumption test. However, for the particular application considered in the present paper, we also need to extend this notion to higher-order clauses sets. Indeed, the most simple notion of analogy, that has to be taken into account in this work, is simply syntactic equality modulo a substitution of the functional or predicate symbols. Hence the notion of generality must combine $H$-subsumption with higher-order matching.

In [5], we have developed an algorithm for checking whether a set of higher-order clauses $S'$ is more general than $S$. This algorithm is based on higher-order AC-unification and quantifier elimination (using the quantifier elimination rules in [3]). Before applying this algorithm, it is interesting to apply the generalization algorithm on the “source problem” $S$ in order to increase the number of sets of clauses “less general” than $S$ (see [4]). However,
the generalization algorithm presented in [4] had some limits: in particular, it is not
colluent, i.e. we can obtain from the same problem $S$ different generalized formulae
$S_1, S_2, \ldots$ that cannot be compared using the generalization ordering. The main aim of
the present work is to give another, more powerful, generalization method based on the
same principle, but overcoming the drawbacks of the previous one. Instead of defining
the generalization algorithm by a set of rewriting rules as done in [4], we show how to compute
the “most general set of formulae” that can be obtained from a given proof. This set of
formulae is described by a set of constrained clauses (with second-order variables replacing
function and predicate symbols) and a second-order unification problem, imposing some
conditions on these second order variables. The second-order unification problem is simply
obtained by considering the conjunction of all the (second-order) unifiere occurring in the
proof. Special kinds of predicate symbols, called domain predicates are added to the
clauses to encode particular instantiations of the variables occurring in these clauses. The
constrained clauses give us a very natural and easy way of formalizing this idea. We show
that this approach is strictly more powerful than our former one.

3.1 A constrained calculus: RAMC

We recall below some necessary notions from [2]. We assume the reader is familiar with
the usual notions in Automated Deduction (clause, formula, interpretation etc.).

**Definition 3** A constrained clause (or a c-clause for short) is a couple $[C : \mathcal{X}]$ where:

- $C$ is a clause (in the standard sense).
- $\mathcal{X}$ is a equational formula (in the usual sense, see for example [3]).

If $C$ is unit then $[C : \mathcal{X}]$ is called a c-literal. If $C$ is empty and $\mathcal{X}$ is satisfiable then
$[C : \mathcal{X}]$ is called an empty clause and denoted by $\Box$.

The refutation rules are simply the standard inference rules (resolution and factorization)
adapted to c-clauses.

**The binary c-resolution** Let $[\neg P(\overline{t}) \lor c_1 : \mathcal{X}]$ and $[P(\overline{s}) \lor c_2 : \mathcal{Y}]$ be two c-clauses
$c_1$ and $c_2$. The rule of binary c-resolution (abbreviated bc-resolution) on $c_1$ and $c_2$ upon
$\neg P(\overline{t})$ and $P(\overline{s})$ is defined as follows:

\[
\frac{[\neg P(\overline{t}) \lor c_1 : \mathcal{X}] \quad [P(\overline{s}) \lor c_2 : \mathcal{Y}]}{[c_1 \lor c_2 : \mathcal{X} \land \mathcal{Y} \land \overline{t} = \overline{s}]}
\]

**The binary c-factorization** The binary c-factorization (abbreviated bc-factorization)
of the c-clause $c = [P(\overline{t}) \lor P(\overline{s}) \lor c' : \mathcal{X}]$ upon $P(\overline{t})$ and $P(\overline{s})$ is defined as follows:

\[
\frac{[P(\overline{t}) \lor P(\overline{s}) \lor c' : \mathcal{X}] \quad [P(\overline{t}) \lor c : \mathcal{X} \land \overline{t} = \overline{s}]}{[P(\overline{s}) \lor c' : \mathcal{X}]}\]

We also use the standard renaming rule.

C-resolution and c-factorisation (and renaming) are sound and refutationally complete
[2]. A sequence of sets of c-clauses $S_1, \ldots, S_n$ is said to be a derivation from $S_1$ if
for all $i \leq n - 1$, $S_{i+1} = S_i \cup \{R_i\}$ where $R_i$ is obtained from c-clauses in $S_i$ by applying
the c-resolution, c-factorization or renaming rules. A derivation $S_1, \ldots, S_n$ is said to be a
refutation of $S_1$ if $S_n$ contains the empty c-clause.
3.2 Generalized c-clause

Generalized c-clause are simply c-clauses in which functional and predicate symbols are replaced by second order variables. More formally:

**Definition 4** A generalized formula is a (higher-order) formula such that any bound variable is of order 1, any free variables is of order at most 2 and any occurrence of a variable $X$ of order 2 occurs in first-order terms of the form $X(t_1, \ldots, t_n)$. A generalized c-clause is of the form $\forall \overline{x}. [C : \forall \overline{x}]$ where $C$ is a generalized clause, $\forall \overline{x}$ is a generalized formula, and $\overline{x}$ a n-tuple of first-order variables.

**Remark 1** Clearly, the problem of checking whether a given generalized formulae is valid or not (and in particular of checking whether a given c-clause is empty) is undecidable. But this is not a problem because this test will be made only of some particular closed instances of these formulae, for which the test is decidable.

The notion of satisfiability is defined as follows: A generalized formula $F$ (resp. set of generalized c-clauses $S$) is said to be satisfiable iff there exists a ground (second-order) substitution $\sigma$ on the free variables of $F$ (resp. $S$) such that $F\sigma$ (resp. $S\sigma$) is satisfiable\(^1\).

Obviously all the notions introduced in Section 3.1, namely the c-resolution and c-factorization rule, the notion of derivation, refutation etc. can be straightforwardly extended to generalized formulae using the same principle.

3.3 Generalization

The generalization method is divided into two steps. The first part consists in an analysis of the proof in order to find its relevant features. These features are expressed by a second-order unification problem. With our constraint-based calculus, it suffices to consider the constraint $\forall \overline{x}$ of the empty clause $[\Box : \forall \overline{x}]$ obtained at the end of the derivation. The second step consists in transforming the unification problem into a simpler form using a set of rewriting rules. These rules do not preserve equivalence hence the obtained problem may have less solutions than the initial one. However, they are sound in the sense that any solution of the obtained problems validates the initial one. This step is not necessary from a theoretical point of view, but allows to simplify the matching process.

These two steps are not clearly distinguished in \cite{4}, where the simplification of the unification problem was implicitly performed during its computation. We believe that this new presentation makes our method much clearer. In the present paper, we focus mainly on the first step, which is the crucial one for proof generalization.

We now give a formal description of the generalization process. A c-clause $C$ is said to be a generalization of a clause $\forall \overline{x} C'$ if $C$ is obtained from $\forall \overline{x} C'$ by replacing each instance of a variable or of a function or predicate symbol by free variables.

**Example.** $P_1(F(A_1), A_2) \forall x_1, x_2. P_2(x_1, x_2)$ is the generalization of $p(f(a), a) \forall x. p(x, x)$.

Let $\Lambda$ be a function from first-order formulae into generalized formula replacing any occurrence of functional, predicate symbols and variables by new, distinct variables of the same type.

\(^1\)By definition of the notion of generalized formula $F\sigma$ must be a first-order formula (after simplification using $\beta$-reduction).
Definition 5 Let $S$ be a set of c-clauses, and $\delta$ a derivation from $S$. The $\Lambda$-generalization of $\delta$ is the sequence $\delta' = \{S'_1, \ldots, S'_n\}$ obtained as follows.

- **Base case.** $S'_1 = \{D_C(x_1, \ldots, x_n) \land \Lambda(C) / C \in S_1\}$, where $D_C$ are new predicate symbols (called domain predicates) and $\{x_1, \ldots, x_n\} = \text{Var}(C)$. Intuitively $D_C$ expresses the conditions on the variables of $C$.

- **Inductive case.**
  - If $S_{i+1}$ is obtained from $S_i$ by applying the c-resolution rule on $[P(\overline{t}) \lor R_1 : \mathcal{X}]$, $[[\neg P(\overline{t})] \lor R_2 : \mathcal{Y}]$, then $S'_{i+1}$ is defined as follows. By construction, there exists in $S'_i$ two c-clauses $[D_1 \lor P'_1 \lor R'_1 \lor \mathcal{X}' \lor \mathcal{Y}]$ and $[D_2 \lor \neg P'_2 \lor R'_2 \lor \mathcal{Y}]$ such that $[[P'_1 \lor R'_1 : \mathcal{X}']$ and $[[\neg P'_2 \lor R'_2 : \mathcal{Y}']$ are generalizations of $[P(\overline{t}) \lor R_1 : \mathcal{X}]$ and $[[\neg P(\overline{t})] \lor R_2 : \mathcal{Y}]$ respectively.
  - The construction is similar if $S_{i+1}$ is obtained by factorization or renaming.

Let $S$ be a set of c-clauses and $\delta = (S_1, \ldots, S_n)$ a refutation of $S$ ($S = S_1$). Let $\delta' = (S'_1, \ldots, S'_n)$ be the $\Lambda$-generalization of $\delta$. By construction, there exists a c-clause $[D : \mathcal{X}] \in S'_i$ where $D$ contains only domain predicates.

Then, the $\Lambda$-c-generalization of $S$ w.r.t. $\delta$ (denoted by $\Theta_{(\delta, \Lambda)}(S)$) is the pair $(S'_i, \mathcal{X}')$, where $\mathcal{X}' = \mathcal{X} \land \bigwedge_{i=1}^{k} D_i = \bot$, where $\{D_1, \ldots, D_k\}$ are the literals of $D$.

**Example 4** Let $S = \{\neg p(x), p(a) \lor p(b)\}$ and $\delta = (S, S \cup \{p(b), \neg p(x')\}, S \cup \{p(b), \neg p(x')\})$. The $\Lambda$-c-generalization of $S$ w.r.t. $\delta$ is the pair $((\forall x. D(x) \lor \neg P_1(x), P_2(A) \lor P_3(B), \mathcal{X})$, where $\mathcal{X} \equiv \exists x_1, x_2. (P_1(x_1) = P_2(A) \land P_2(x_2) = P_3(B) \land D(x_1) = \bot \land D(x_2) = \bot)$.

The following theorem express of soundness of the generalization process.

**Theorem 1** Let $S$ be a set of c-clauses and $\delta$ a derivation of $S$. Let $(S', \mathcal{X}) = \Theta_{(\delta, \Lambda)}(S)$. For any ground solution $\sigma$ of $\mathcal{X}\sigma$, $S'\sigma$ is unsatisfiable. Moreover the refutation of $S$ can be efficiently transformed into a refutation of $S'\sigma$.

A set of c-clauses $S'$ is said to be an instance of $\Theta_{(\delta, \Lambda)}(S)$ iff there exists $\sigma$ such that $S\sigma \geq_{\text{gen}} S'$ and $\mathcal{X}\sigma \equiv \top$. The pair $\Theta_{(\delta, \Lambda)}(S)$ is in some sense the “most general formula” that can be obtained from $\delta$, because each set of c-clauses $S'$ having the “same proof structure” as $\delta$ must be an instance of $\Theta_{(\delta, \Lambda)}(S)$.

### 3.4 The analogy detection step

Let $S$ be a set of c-clauses and $\delta$ a derivation of $S$. Let $(S', \mathcal{X}) = \Theta_{(\delta, \Lambda)}(S)$. Let $S_T$ a new set of c-clauses (the target formula). We want to use our generalization algorithm for finding the refutation of $S_T$ from the refutation $\delta$.

According to Theorem 1, it suffices to use the matching algorithm in order to check whether the proposed conjecture is an instance of the initial formulae $S'$ and to check that the constraint part of the conclusion is satisfiable (which is decidable, since it is a first-order formulae). More precisely, we have to find a substitution $\sigma$ such that $S\sigma \geq_{\text{gen}} S_T$ and $\mathcal{X}\sigma \equiv \top$. This can be done using specific algorithms based on higher-order unification modulo AC and quantifier elimination. This part will not be explained in the present paper (see [6, 5] for details).
4 Combining the generalization algorithm with function introduction

4.1 The function introduction rule

The function introduction rule has been defined in [1], and further investigated in [7, 8]. The aim of the rule is to introduce new functional symbols in the clauses set. These functional symbols are introduced by using a distribution of quantifiers rule followed by a skolemization step. In [1], it is shown that adding this rule to the resolution calculus can significantly shorten the obtained proofs.

Let us give an example. Consider the following clause \( C = \forall x, y. A(x, y) \lor B(x, y) \). Obviously the formula \( \forall x (\forall y. A(x, y)) \lor (\exists y. B(x, y)) \) is a logical consequence of \( C \). After skolemization we get \( \forall x, y. A(x, y) \lor B(x, f(x)) \). This new clause can be added to the initial set of clauses.

What is interest of using function introduction? Simply that the obtained set may be far easier to prove than the original one. Indeed, applying function introduction reduces the number of clauses that are shared by two given literal (here \( A(x, y) \) and \( B(x, f(x)) \) share only one variable, whereas \( A(x, y) \) and \( B(x, y) \) share two variables). This can allow in some case to apply splitting rule. Moreover, it is shown in [1] that this can also result in a non-elementary shortening of the proof. We recall below the definition of the function introduction rule (see [1] for details).

**Definition 6** Let \( \forall x_1, \ldots, x_m. C \) be a clause. If \( C_1, C_2 \neq \emptyset \) and \( C \equiv C_1 \lor C_2 \) then \( F_i(\forall x_1, \ldots, x_m. C) = \forall x_1, \ldots, x_j(\forall y_1, \ldots, y_k. C_1) \lor \forall z_1, \ldots, z_l. C_2 \), where \( \{x_1, \ldots, x_j\} = \text{Var}(C_1) \cap \text{Var}(C_2) \), \( \{y_1, \ldots, y_k\} = \text{Var}(C_1) \setminus \text{Var}(C_2) \) and \( \{z_1, \ldots, z_l\} = \text{Var}(C_2) \setminus \text{Var}(C_1) \) is a reduced f-form of \( C \). For a clause set \( C = \{C_1, \ldots, C_n\} \), let \( F_i(C) = \bigwedge_{i=1}^n F_i(C_i) \).

Let \( C \) be a set of clauses and \( C \in C \). Let \( F_i(C) \equiv \forall x_1, \ldots, x_m. (F_1 \lor F_2) \) be a reduced f-form of \( C \) and let \( \{u_1, \ldots, u_k\} \subseteq \{x_1, \ldots, x_m\} \) and \( \{v_1, \ldots, v_l\} = \{x_1, \ldots, x_m\} \setminus \{u_1, \ldots, u_k\} \). Then the clause form of \( F_i(C) \land \forall v_1, \ldots, v_l. Q_1 u_1 \ldots Q_k u_k \land F_1 \lor Q_1^2 u_1 \ldots Q_k^2 u_k F_2 \) where \( Q_i \in \{\forall, \exists\} \) for \( i = 1, 2, \ldots, n \) and \( Q_i^2 \) dual to \( Q_i \), is called an F-extension of \( C \).

We say the terms \( t_u, \ldots, t_k \) used for replacing the variables \( u_1, \ldots, u_k \) in the Skolemization step are Skolem term corresponding to \( u_1, \ldots, u_k \).

4.2 Combining F-extension with generalization of proofs

F-extension is a powerful way of reducing proof size. It can be applied during proof search, as an inference rule. However it can also be used after proof search in order to impose structure on resolution proofs and therefore the improve the readability of the proof [9]. Applying the function introduction rule allows to eliminate some redundant (in some sense) subpart of the proof which both decreases the size of the proof and improves it readability. Therefore a very natural question arises: can the F-extension be combined with our generalization algorithm? In the following, we give an positive answer to this problem.

The new generalization algorithm

The main difficulty involved by the F-extension rule is that the condition on the premisses of the rule are not unification problems as for the previous rules (resolution and factorization). Therefore, the techniques used in Section 3.3 for generalizing the refutation are
no more sufficient. This problem will be solved by adding some further conditions on
the domain predicates, insuring that the condition on the variables expressed by these
predicate satisfies some properties allowing to apply the $F$-extension rule. More precisely,
we modify the generalization algorithm as follows.

- We add 3 new inference rules. The first one is $F$-extension, and the other ones are
  constraint simplification rules aiming at transforming $c$-clauses into standard clauses.
  They have to be applied prior to the $F$-extension rule.

**Decomposition.**

\[
\frac{\exists C : F(\overline{t}) = G(\overline{\sigma}) \land \mathcal{X}}{\exists C : \overline{t} \land \mathcal{X} \land F = G}
\]

**Replacement.**

\[
\forall \overline{x} . \exists C : x = t \land \mathcal{X}
\]

\[
\forall \overline{x} . \exists C \{ x \rightarrow t \} : \mathcal{X} \{ x \rightarrow t \}
\]

- **Construction of the generalized derivation.** Applications of the Decomposition and
  Replacement rules are generalized as usual. The $F$-extension rule is generalized as
  follows.

Let $C$ be a set of clauses and $C \in C$. Let

\[
C \equiv \forall x_1, \ldots, x_m (\forall \overline{y} F_1 \lor \forall \overline{y} F_2) \rightarrow_{F-\text{extension}} \forall v_1, \ldots, v_l Q_1 u_1 \ldots Q_k u_k \forall \overline{y} \exists \overline{z} F_1 \lor Q_1^d u_1 \ldots Q_k^d u_k \forall \overline{z} F_2
\]

be an application of the $F$-extension rule (see Definition 6 for notations).

Let $e = (t_1, \ldots, t_n)$ be a tuple and $I \subseteq [1..n]$. We denote by $e_I$ the subsequence of
$e$ containing only elements of indice $i \in I$.

By construction, there exists a clause $\forall x_1, \ldots, x_m, \exists \overline{y}, \exists \overline{z}. D \lor F'_1 \lor F'_2$ of $C$ in the set
of $c$-clauses, where $F'_1, F'_2$ are generalization of $F_1, F_2$, respectively. We deduce:

\[
\forall v_1, \ldots, v_l Q_1 u_1 \ldots Q_k u_k F'_1 \lor \forall \overline{y} F'_1 \lor D_1 \lor D'_1 \lor Q_1^d u_1 \ldots Q_k^d u_k \forall \overline{y} F'_2 \lor D_2 \lor D'_2,
\]

with the constraints:

- $D'$ is the set of literals in $D$ containing only variables in $\{x_1, \ldots, x_m\}$;
- $D_1$ is the set of literals in $D$ containing only variables in $\overline{y}$;
- $D_2$ is the set of literals in $D$ containing only variables in $\overline{z}$;
- $d_1, \ldots, d_j$ are the domain predicates occurring in $D \setminus (D_1 \cup D_2 \cup D')$. $e_i'$ are the
  indices of the argument of $d_i$ containing only variables in $\{x_1, \ldots, x_m\} \cup \overline{y}$; $e_i''$ are the
  indices of the argument of $d_i$ containing only variables of $\{x_1, \ldots, x_m\} \cup \overline{z}$.
- $D'_1 = \forall_{d_i(t) \in D_1 \land 1 \leq i \leq n} d_i(t|e'_i)$.
- $D'_2 = \forall_{d_i(t) \in D_2 \land 1 \leq i \leq n} d_i(t|e''_i)$.

Adding these new constraints on $d_1, \ldots, d_j$ insures that the generalized clause is
decomposable, which makes the application of the $F$-extension rule possible.

Clearly, the obtained formula must be skolemized in order to eliminated existential
quantifiers. For doing that, it suffices to generalized to skolemized normal form i.e.
to replace each existential variable $x$ in the generalized formula by a term of the
form $X(x_1, \ldots, x_n)$ (where $x_1, \ldots, x_n$ is the set of free variables in the existential
subformula). Moreover, we also have to add the $\forall \overline{F}$ on top of the constraints of the
obtained generalized derivation (in order to insure that the functional variables in
the derivation cannot be instanciated by skolem functions).
Example 5 We consider the following derivation:

\[ p(f(x, z) \lor \neg q(x)) \lor q(z) \]

\[ \neg q(f(x', y') \lor r(x')) \]

\[ \neg q(c_1) \lor \neg q(c_2) \]

\[ \neg r(\bar{z}) \]

\[ q(x) \lor r(x') : f(x, x) = f(x', y') \]

\[ [q(x) \lor r(x') : x = x' \land x = y'] \] (decomposition)

\[ q(x) \lor r(x) : \top \]

\[ q(x) \lor r(c) : \top \]

\[ [\neg q(c_2) \lor r(c) : x = c_1] \]

\[ [q(x'') \lor r(c) : \top] \]

\[ [r(c) : x = c_1 \land x'' = c_2] \]

\[ [\top : x = c_1 \land x'' = c_2 \land z = c] \]

The first part of the refutation (before the \( F \)-extension step) is generalized as follows:

\( (F_1, F_2, C_1, C_2 \) denotes function variables (of arity 2, 2, 0, 0 respectively), \( D_1, D_2, D_3, D_4 \) denotes the domain predicates).

\[ \forall x_1, x_2, x_3, D_1(x_1, x_2, x_3) \lor p(F_1(x_1, x_2)) \lor q(x_3) \] (given)

\[ \forall x_1, x'_2, y'_1, D_2(x'_1, x'_2, y'_1) \lor \neg p(F_2(x'_1, y'_1)) \lor r(x'_2) \] (given)

\[ \neg q(c_1) \lor \neg q(c_2) \] (given)

\[ \forall z_2, D_3(z_1) \lor \neg r(z) \] (given)

\[ \forall x_1, x_2, x_3, x'_1, x'_2, y'_1, D_1(x_1, x_2, x_3) \lor D_2(x'_1, x'_2, y'_1) \lor q(x_3) \lor r(x'_2) : F_1(x_1, x_2) = F_2(x'_1, y'_1) \]

\[ \forall x_1, x_2, x_3, x'_1, x'_2, y'_1, D_1(x_1, x_2, x_3) \lor D_2(x'_1, x'_2, y'_1) \lor q(x_3) \lor r(x'_2) : F_1 = F_2 \land x_1 = x'_1 \land x_2 = y'_1 \]

\[ \forall x_1, x_2, x_3, x'_1, x'_2, x'_3, D_4(z_1) \lor D_3(x_1, x'_2, x'_3) \lor q(x_3) \lor r(x'_2) : F_1 = F_2 \] (given)

Now we have to generalize the \( F \)-extension step. We have to make sure that the clause is decomposable. Here it is clearly the case, since the tuples of variables \( \bar{y} \) and \( \bar{z} \) (with the above notation, see the definition of the \( F \)-extension rule) are empty. Hence, the \( F \)-extension rule is applicable on the generalized formula, and we simply obtain the following generalized c-clause:

\[ \forall x_1, x_2, x_3, x'_1, x'_2 \left[ D_1(x_1, x_2, x_3) \lor D_2(x_1, x'_2, x_2) \lor q(x_3) \lor r(C) : F_1 = F_2 \right] \]

Due to space restrictions, we do not give the generalized of the rest of the derivation. The obtained generalized refutation is the couple \((S', \mathcal{X})\) where \( S' \) is the set of generalized clauses 1 - 4

and

\[ \mathcal{X} \equiv \forall C \exists x_1, x_2, x_3, x'_1, x'_2 (D_1(x_1, x_2, x_3) \lor D_2(x_1, x'_2, x_2) \lor D_1(x_1, x_2, x_3) \lor D_2(x_1, x'_2, x_2) \lor D_4(z_1)) = \bot \land F_1 = F_2 \land x_3 = C_1 \land x'_3 = C_2 \land z_1 = C. \]

In the following example, we show what happen if \( \bar{y} \) and \( \bar{z} \) are not empty.

Example 6 Now, we replace in the previous example the first two clauses by \( p(f(x, x)) \lor q(x, u) \) and \( p(f(x', y')) \lor q(x', v) \) respectively. Clause 7 becomes: \([q(x, u) \lor r(x, v) : \top]\].

By applying \( F \)-extension rule we obtain: \([q(x, u) \lor r(c, v) : \top]\]. Hence the generalized clause 7 becomes:

\[ \forall x_1, x_2, x_3, x'_1, u_1, v_1, D_1(x_1, x_2, x_3, u_1) \lor D_2(x_1, x'_2, x_2, v_1) \lor q(x_3, u_1) \lor r(x'_2, v_1) : F_1 = F_2 \]

Here, we have to impose further conditions on the domain predicates \( D_1, D_2 \), since their arguments contain simultaneously variables from \( u_1, v_1 \) and from \( x_1, x_2, x_3, x'_2 \). According to the previous definition, we obtain the following conditions \( D_1 = \lambda z_1, z_2, z_3, z_4. (D'_1(z_1, z_2, z_3) \lor D'_1(z_4)) \) and \( D_2 = \lambda z_1, z_2, z_3, z_4. (D'_2(z_1, z_2, z_3) \lor D'_2(z_4)) \)
where \(D'_1, D''_1, D'_2, D''_2\) are new variables. These conditions ensure that the clause is decomposable which makes the application of the \(F\)-extension possible. Without these decomposability, the quantifiers \(\forall z_1\) and \(\forall v_1\) could not be shifted on the literals \(q(x_3, u_1)\) and \(r(x'_2, v_1)\) because \(z_1\) and \(v_1\) would occur in literals containing variables from \(x_1, x_2, x_3, x'_2\).

We obtain the clause
\[
\forall x_1, x_2, x_3, x'_2, u_1, v_1, D'_1(x_1, x_2, x_3) \lor D_2(x_1, x'_2, x_2) \lor D''_1(u_1) \lor q(x_3, u_1) \lor D''_2(v_1) \lor r(x'_2, v_1)
\]
with the constraints:
\[
D_1 = \lambda z_1, z_2, z_3, z_4. (D'_1(z_1, z_2, z_3) \lor D''_1(z_4)) \land D_2 = \lambda z_1, z_2, z_3, z_4. (D_2(z_1, z_2, z_3) \lor D''_2(z_4)) \land F_1 = F_2
\]

For any refutation \(\delta\) of \(S\), we denote by \(\mathcal{G}^{ext}_{(\delta, \lambda)}(S)\) the pair \((S', \lambda)\) obtained from \(\delta\) by using the extended generalization algorithm. Then the following theorem ("analogous" to Theorem 1) holds:

**Theorem 2** Let \(S\) be a set of \(c\)-clauses and \(\delta\) a refutation of \(S\). Let \((S', \lambda) = \mathcal{G}^{ext}_{(\delta, \lambda)}(S)\). For any ground solution \(\sigma\) of \(\lambda \sigma\), \(S' \sigma\) is unsatisfiable. Moreover the refutation of \(S\) can be efficiently transformed into a refutation of \(S' \sigma\).

These results allow to combine \(F\)-extension with proof generalization. In particular, the lemma generation technique presented in [9] can be applied before proof generalization in order to get a shorter and more structured proof. However, it must be emphasized that this process is not monotonic, i.e., we can get a generalized proof that is not more general than the one that would have been obtained without using the \(F\)-extension rule. But this is compensated by the advantages of function introduction for structuring proofs.

**References**


181