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 readibility. 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 been 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

proofs, 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 $A \equiv \forall k.\Sigma_{i=0}^{k-1} 2 \times i + 1 = k^2$ and $B \equiv \forall k.\exists x.\Sigma_{i=0}^{k-1} 2 \times i + 1 = x^2$. 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, $F_1 : \forall y.B(y) \vee \top$ will not be said to be more general than $F_2 : \exists x.A(x) \vee \forall y.\neg A(y)$ 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 commun".

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, instanciating 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) $\forall y.(P(y) \vee \exists x.\neg P(x))$, whereas the obtained set of clauses after generalization corresponds to $\forall y.(\neg P(y) \vee P(y))$. The latter is obviously more general than the former since an existential quantified variable has been instanciated: $\forall y.(P(y) \vee \exists x.\neg P(x))$ only says that there exists an x such that $P(x)$, whereas the the formula $\forall y.(\neg P(y) \vee P(y))$ gives explicitely the value of the variable x (here $x = y$).

Example 1 *Let* $A \equiv \{A(b), \neg A(b) \vee B, \neg B \vee \neg C, C\}$ *and* $B \equiv \{\forall x.A(x), \forall x.\neg A(x) \vee B, \neg B, C\}$.

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, is such a way 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:* $\forall x.P(x), \neg P(a) \rightarrow \square$ *(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:* $A' \equiv \Sigma_{i=0}^{k-1} 2 \times i + 1 \neq k^2$ *and* $B' \equiv \forall x.\Sigma_{i=0}^{k-1} 2 \times i + 1 \neq x^2$ *(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 instanciated 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 $S >_{\text{gen}} S'$, 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* confluent, 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) unifiers occurring in the proof. Special kinds of predicate symbols, called *domain predicates* are added to the clauses to encode particular instanciations 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* $\square$.

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

The binary c-resolution Let $[\![\neg P(\overline{t}) \vee c'_1 \ : \mathcal{X}]\!]$ and $[\![P(\overline{s}) \vee c'_2 \ : \mathcal{Y}]\!]$ 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:

$$\frac{[\![\neg P(\overline{t}) \vee c'_1 \ : \mathcal{X}]\!] \qquad [\![P(\overline{s}) \vee c'_2 \ : \mathcal{Y}]\!]}{[\![c'_1 \vee c'_2 \ : \mathcal{X} \wedge \mathcal{Y} \wedge \overline{t} = \overline{s}]\!]}$$

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

$$\frac{[\![P(\overline{t}) \vee P(\overline{s}) \vee c' \ : \mathcal{X}]\!]}{[\![P(\overline{s}) \vee c' \ : \mathcal{X} \wedge \overline{t} = \overline{s}]\!]}$$

We also use the standard renaming rule.

C-resolution and c-factorisation (and renaming) are sound and refutationaly complete [2]. A sequence of sets of c-clauses $S_1, \ldots, S_n$ is said to be a *derivation from* S_1 iff 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 iff 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* $\forall \overline{x}.[\![C \ : \mathcal{X}]\!]$ *where* C *is a generalized clause,* $\mathcal{X}$ *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* $\mathcal{F}$ (resp. set of generalized c-clauses S) is said to be *satisfiable* iff there exists a ground (second-order) substitution σ on the free variables of $\mathcal{F}$ (resp. S) such that $\mathcal{F}\sigma$ (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 $\mathcal{X}$ of the empty clause $[\![\Box \ : \mathcal{X}]\!]$ 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 [4], where the simplification of the unification problem was implicitely 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 C' iff C is obtained from C' by replacing each instance of a variable or of a function or predicate symbol by free variables.

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

Let Λ 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 $\mathcal{F}\sigma$ must be a first-order formula (after simplification using β-reduction).

Definition 5 *Let S be a set of c-clauses, and δ a derivation from S. The Λ-generalization of δ is the sequence $\delta' = \{S'_1, \ldots, S'_n\}$ obtained as follows.*

- **Base case.** *$S'_1 = \{D_C(x_1, \ldots, x_n) \vee \Lambda(C)/C \in S_1\}$, where D_C are new predicate symbols (called* domain predicates*) and $\{x_1, \ldots, x_n\} = \mathcal{V}ar(C)$. 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 $[\![P(\overline{t}) \vee R_1 \ : \mathcal{X}]\!]$, $[\![\neg P(\overline{s}) \vee R_2 \ : \mathcal{Y}]\!]$, then S'_{i+1} is defined as follows. By construction, there exists in S'_i two c-clauses $[\![D_1 \vee P'_1 \vee R'_1 \ : \mathcal{X}']\!]$ and $[\![D_2 \vee \neg P'_2 \vee R'_2 \ : \mathcal{Y}']\!]$ such that $[\![P'_1 \vee R'_1 \ : \mathcal{X}']\!]$ and $[\![\neg P'_2 \vee R'_2 \ : \mathcal{Y}']\!]$ are generalization of $[\![P(\overline{t}) \vee R_1 \ : \mathcal{X}]\!]$ and $[\![\neg P(\overline{s}) \vee R_2 \ : \mathcal{Y}]\!]$ respectively.*
    $S'_{i+1} = S'_i \cup \{[\![D_1 \vee D_2 \vee R'_1 \vee R'_2 \ : \mathcal{X}' \wedge \mathcal{Y}' \wedge P'_1 = P'_2]\!]\}$.
  - *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 Λ-generalization of δ. By construction, there exists a c-clause $[\![D \ : \mathcal{X}]\!] \in S'_n$ where D contains only domain predicates.

Then, the *Λ-c-generalization* of S w.r.t. δ (denoted by $\mathfrak{G}_{(\delta,\Lambda)}(S)$) is the pair $(S'_1, \mathcal{X}')$, where $\mathcal{X}' = \mathcal{X} \wedge \bigwedge_{i=1}^{k} D_i = \bot$, where $\{D_1, \ldots, D_k\}$ are the literals of D.

Example 4 *Let $S = \{\neg p(x), p(a) \vee p(b)\}$ and $\delta = (S, S \cup \{p(b)\}, S \cup \{p(b), \neg p(x')\}, S \cup \{p(b), \neg p(x'), \Box\})$. The Λ-c-generalization of S w.r.t. δ is the pair*

$$(\{\forall x.D(x) \vee \neg P_1(x), P_2(A) \vee P_3(B)\}, \mathcal{X})$$

, where $\mathcal{X} \equiv \exists x_1, x_2.(P_1(x_1) = P_2(A) \wedge P_2(x_2) = P_3(B) \wedge D(x_1) = \bot \wedge D(x_2) = \bot)\}$.

The following theorem express of soundness of the generalization process.

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

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

3.4 The analogy detection step

Let S be a set of c-clauses and δ a refutation of S. Let $(S', \mathcal{X}) = \mathfrak{G}_{(\delta,\Lambda)}(S)$. 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 δ.

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 σ such that $S\sigma >_{\text{gen}} S_T$ 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 $C = \forall x, y.A(x, y) \vee B(x, y)$. Obviously the formula $\forall x.(\forall y.A(x, y)) \vee (\exists y.B(x, y))$ is a logical consequence of C. After skolemization we get $\forall x, y.A(x, y) \vee B(x, f(x))$. 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 same 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 $C \equiv C_1 \vee C_2$ then $F_\tau(\forall x_1, \ldots, x_m.C = \forall x_1, \ldots, x_j(\forall y_1, \ldots, y_k.C_1) \vee \forall z_1, \ldots, z_l.C_2)$, where $\{x_1, \ldots, x_j\} = \mathcal{V}ar(C_1) \cap \mathcal{V}ar(C_2)$, $\{y_1, \ldots, y_k\} = \mathcal{V}ar(C_1) \setminus \mathcal{V}ar(C_2)$ and $\{z_1, \ldots, z_l\} = \mathcal{V}ar(C_2) \setminus \mathcal{V}ar(C_1)$ is a* reduced f-form *of C. For a clause set $\mathcal{C} = \{C_1, \ldots, C_n\}$, let $F_\tau(\mathcal{C}) = \bigwedge_{i=1}^{n} .F_\tau(C_i)$.*

Let $\mathcal{C}$ be a set of clauses and $C \in \mathcal{C}$. Let $F_\tau(C) \equiv \forall x_1, \ldots, x_m.(F_1 \vee F_2)$ 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 $F_\tau(\mathcal{C}) \wedge \forall v_1, \ldots, v_l.Q_1 u_1 \ldots Q_k u_k.F_1 \vee Q_1^d u_1 \ldots Q_k^d u_k F_2$ where $Q_i \in \{\forall, \exists\}$ for $i = 1, 2, \ldots, n$ and Q_i^d dual to Q_i, is called an F-extension of $\mathcal{C}$.

We say the terms $t_{u_1}, \ldots, t_{u_k}$ 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 *readibility* 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 readibility. 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.

  $$\frac{[\![C \ : F(\overline{t}) = G(\overline{s}) \wedge \mathcal{X}]\!]}{[\![C \ : \overline{t} = \overline{s} \wedge \mathcal{X} \wedge F = G]\!]}$$

  Replacement.

  $$\frac{\forall x.[\![C \ : x = t \wedge \mathcal{X}]\!]}{\forall x.[\![C\{x \to t\} \ : \mathcal{X}\{x \to t\}]\!]}$$

- **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 $\mathcal{C}$ be a set of clauses and $C \in \mathcal{C}$. Let

  $C \equiv \forall x_1, \ldots, x_m.(\forall \overline{y}.F_1 \vee \forall \overline{z}.F_2) \rightarrow_{F\text{-extension}} \forall v_1, \ldots, v_l.Q_1u_1 \ldots Q_ku_k.\forall \overline{y}.F_1 \vee Q_1^d u_1 \ldots Q_k^d u_k \forall \overline{z}.F_2$

  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 $\forall x_1, \ldots, x_m, \overline{y}, \overline{z}.D \vee F_1' \vee F_2'$ of C in the set of c-clauses, where F_1', F_2' are generalization of F_1, F_2, respectively. We deduce:

  $\forall v_1, \ldots, v_l.Q_1u_1 \ldots Q_ku_k.D' \vee \forall \overline{y}.F_1' \vee D_1 \vee D_1' \vee Q_1^d u_1 \ldots Q_k^d u_k \forall \overline{z}.F_2 \vee D_2 \vee D_2'$, with the constraints: $\bigwedge_{i=1}^{j}.d_i = \lambda x_1, \ldots, x_{n_i}.d_i'(x_j/j \in e_i') \vee d_i''(x_j/j \in e_i'')$ where:

  - 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 $\{x_1, \ldots, x_m\} \cup \overline{z}$.
  - $D_1' = \bigvee_{d_i(t) \in D, 1 \leq i \leq n} d_i'(t_{|e_i'})$.
  - $D_2' = \bigvee_{d_i(t) \in D, 1 \leq i \leq n} d_i''(t_{|e_i''})$.

  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 generalizedd 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:*

1 $p(f(x,x)) \vee q(x)$ (given)
2 $\neg p(f(x',y')) \vee r(x')$ (given)
3 $\neg q(c_1) \vee \neg q(c_2)$ (given)
4 $\neg r(z)$ (given)
5 $[\![q(x) \vee r(x') \ : f(x,x) = f(x',y')]\!]$ (resolution 1, 2)
6 $[\![q(x) \vee r(x') \ : x = x' \wedge x = y']\!]$ (decomposition 5)
7 $[\![q(x) \vee r(x) \ : \top]\!]$ (replacement)
8 $[\![q(x) \vee r(c) \ : \top]\!]$ (F-extension)
9 $[\![\neg q(c_2) \vee r(c) \ : x = c_1]\!]$ (resolution)
10 $[\![q(x'') \vee r(c) \ : \top]\!]$ (renamming)
11 $[\![r(c) \vee r(c) \ : x = c_1 \wedge x'' = c_2]\!]$ (resolution)
12 $[\![r(c) \ : x = c_1 \wedge x'' = c_2]\!]$ (factorization)
13 $[\![\Box \ : x = c_1 \wedge x'' = c_2 \wedge z = c]\!]$ (resolution)

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).

1 $\forall x_1, x_2, x_3. D_1(x_1, x_2, x_3) \vee p(F_1(x_1, x_2)) \vee q(x_3)$ (given)
2 $\forall x'_1, x'_2, y'_1. D_2(x'_1, x'_2, y'_1) \vee \neg p(F_2(x'_1, y'_1)) \vee r(x'_2)$ (given)
3 $\neg q(C_1) \vee \neg q(C_2)$ (given)
4 $\forall z_1. D_4(z_1) \vee \neg r(z)$ (given)
5 $\forall x_1, x_2, x_3, x'_1, x'_2, y'_1. [\![D_1(x_1, x_2, x_3) \vee D_2(x'_1, x'_2, y'_1) \vee q(x_3) \vee r(x'_2) \ : F_1(x_1, x_2) = F_2(x'_1, y'_1)]\!]$
6 $\forall x_1, x_2, x_3, x'_1, x'_2, y'_1. [\![D_1(x_1, x_2, x_3) \vee D_2(x'_1, x'_2, y'_1) \vee q(x_3) \vee r(x'_2) \ : F_1 = F_2 \wedge x_1 = x'_1 \wedge x_2 = y'_1]\!]$
7 $\forall x_1, x_2, x_3, x'_2. [\![D_1(x_1, x_2, x_3) \vee D_2(x_1, x'_2, x_2) \vee q(x_3) \vee r(x'_2) \ : F_1 = F_2]\!]$

Now we have to generalize the F-extension step. We have to make sure that the clause is decomposable. Here is it clearly the case, since the tuples of variables $\overline{y}$ and $\overline{z}$ (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:

$$\forall x_1, x_2, x_3, x'_2. [\![D_1(x_1, x_2, x_3) \vee D_2(x_1, x'_2, x_2) \vee q(x_3) \vee r(C) \ : F_1 = F_2]\!]$$

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

$\mathcal{X} \equiv \forall C. \exists x_1, x_2, x'_3, x_3, x'_2. (D_1(x_1, x_2, x'_3) \vee D_2(x_1, x'_2, x_2) \vee D_1(x_1, x_2, x_3) \vee D_2(x_1, x'_2, x_2) \vee D_4(z_1)) = \bot \wedge F_1 = F_2 \wedge x_3 = C_1 \wedge x'_3 = C_2 \wedge z_1 = C.$

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

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

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

$\forall x_1, x_2, x_3, x'_2, u_1, v_1. [\![D_1(x_1, x_2, x_3, u_1) \vee D_2(x_1, x'_2, x_2, v_1) \vee q(x_3, u_1) \vee r(x'_2, v_1) \ : F_1 = F_2]\!]$

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 $D_1 = \lambda z_1, z_2, z_3, z_4. (D'_1(z_1, z_2, z_3) \vee D''_1(z_4))$ and $D_2 = \lambda z_1, z_2, z_3, z_4. (D'_2(z_1, z_2, z_3) \vee D''_2(z_4))$*

where D'_1, D''_1, D'_2, D''_2 *are new variables. These conditions insure that the clause is decomposable which makes the application of the* F*-extension possible. Without these condition, 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

$\forall x_1, x_2, x_3, x'_2, u_1, v_1. D'_1(x_1, x_2, x_3) \vee D_2(x_1, x'_2, x_2) \vee D''_1(u_1) \vee q(x_3, u_1) \vee D''_2(v_1) \vee r(x'_2, v_1)$

with the constraints:

$D_1 = \lambda z_1, z_2, z_3, z_4.(D'_1(z_1, z_2, z_3) \vee D''_1(z_4)) \wedge D_2 = \lambda z_1, z_2, z_3, z_4.(D'_2(z_1, z_2, z_3) \vee D''_2(z_4)) \wedge F_1 = F_2$

For any refutation δ of S, we denote by $\mathfrak{G}^{ext}_{(\delta,\Lambda)}(S)$ the pair $(S', \mathcal{X})$ obtained from δ 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* δ *a refutation of* S*. Let* $(S', \mathcal{X}) = \mathfrak{G}^{ext}_{(\delta,\Lambda)}(S)$*. For any ground solution* σ *of* $\mathcal{X}\sigma$*,* $S'\sigma$ *is unsatisfiable. Moreover the refutation of* S *can be efficently 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

[1] M. Baaz and A. Leitsch. Complexity of resolution proofs and function introduction. *Annals of Pure and Applied Logic*, 20:181–215, 1992.

[2] R. Caferra and N. Zabel. A method for simultaneous search for refutations and models by equational constraint solving. *Journal of Symbolic Computation*, 13:613–641, 1992.

[3] H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–475, 1989.

[4] G. Défourneaux, C. Bourely, and N. Peltier. Semantic generalizations for proving and disproving conjectures by analogy. *Journal of Automated Reasoning*, 20(1 & 2), 1998. Special issue of JELIA's best papers.

[5] G. Défourneaux and N. Peltier. Analogy and abduction in automated reasoning. In M. E. Pollack, editor, *Proceedings of IJCAI'97, Nagoya, Japan*, August 23–29 1997.

[6] G. Défourneaux and N. Peltier. Partial matching for analogy discovery in proofs and counter-examples. In W. McCune, editor, *Proceedings of CADE 14*. Springer, July 1997. LNAI 1249.

[7] U. Egly. Shortening proofs by quantifier introduction. In *Proceedings of LPAR 92*, pages 148–159. Springer, 1992. LNAI 624.

[8] U. Egly. On different concepts of function introduction. In *Computational Logic and Proof Theory, KGC 93*, pages 172–183. Springer, LNCS 713, 1993.

[9] K. Hörwein. Structuring resolution proofs by introducing new lemmata (or how to improve the readability of atp-generated proofs). *Journal of Automated Reasoning*, 19:173–203, 1997.