Proof Analogy In Interactive Theorem Proving:
A Method To Express And Use It Via Second Order Pattern Matching
Thierry Boy de la Tour and Ricardo Caferra
LIFIA, BP 68, 38402 St Martin d'Hères Cedex, France
Telex: USMG 980 134F

Abstract

A method is presented to express and use syntactic analogies between proofs in interactive theorem proving and proof checking. Up to now, very few papers have addressed instances of this problem. The paradigm of "proposition as types" is adopted and proofs are represented as terms. The proposed method is to transform a known proof of a theorem into what might become a proof of an "analogous" — according to the user's proposition, namely the one to be proved. This transformation is expressed by means of second order pattern matching (this may be seen as a generalization of rewriting rules), thus allowing the use of variable function symbols. For the moment, it is up to the user to discover the transformation rule, and the paper deals only with the problem of managing it. We explain the proposed analogy treatment with a fully developed running example.

In looking for a proof of a theorem it is very helpful to find "analogies" with proofs of already proved theorems in order to guide the discovery of the new proof.

A typical example which can be found in mathematical texts is the statement "this theorem can be proved as the previous one". This sentence stands for a proof which is analogous to the designated one. But much more "analogy information" may be conveyed by the text. Actually, a larger amount of information seems to be needed if mechanization is considered.

Many questions are raised by these simple intuitive observations, the more important are:

1. What does "analogy" mean in this context?
2. How to formalize in some way this analogy (especially in mechanized theorem proving or proof checking)?
3. What is the proof representation adapted to this notion of analogy?
4. At which level of abstraction is analogy useful and manageable?
5. Are we interested in syntactic, or semantic analogies or both?
6. Are there tools adapted to the handling of this notion of analogy?
7. Is it interesting to modify (or extend) these tools in order to make them more powerful for the treatment of analogy understood with the adopted meaning?

In this paper an attempt is made to partially answer most of these questions:

- Obviously, we cannot answer the first question in any general way, but we propose some kind of "syntactic analogy" and we leave (by now) to the user the task of discovering "analogy". A high level language and some flexibility to formalize (with constraints) these analogies in a nondeterministic way are offered to him.
- The user must formalize the analogies as second order transformation rules corresponding to the transformation from a proof to what is considered (by the user) as an analogous one.
- We may consider as analogous proofs in a large spectra with two ends: proofs are analogous just because they are proofs or just because they are the same. But these kinds of analogy — too much or not enough general— are useless. The adopted analogy must therefore not reach one or the other of these ends. We have chosen to emphasize analogies on the proofs structure.
- We have decided to adopt the so-called "proposition as types" paradigm, and thus represent a proof as a term the type of which is the proposition being proved.
- We consider that a second order pattern matching algorithm is a good tool to be used in a first approach to syntactical analogy.
- Only syntactic analogies are manageable with the chosen tool.

From: AAAI-87 Proceedings. Copyright ©1987, AAAI (www.aaai.org). All rights reserved.
Some modification of Huet's second order pattern matching is currently studied.

To our knowledge analogy has been considered in theorem proving in very few papers (see for example the pioneer work [Kling, 1971] and also [Plaisted, 1981] for the use of abstraction in the resolution method) and we do not know about papers treating analogy in a "proof as term" approach, which is the one chosen in the present work. In [Constable et al, 1985] it is suggested that

\[
\begin{align*}
\text{one can imagine writing very general "transformation tactics" (for details see [Constable et al, 1985]) to construct proofs by analogy to existing proofs} \end{align*}
\]

but no indication is given about how to tackle this problem.

The structure of the paper is as follows:

In section 2 we present some generalities about analogy and explain why we have chosen second order transformation rules. In section 3 we expose the notion of proof as term and introduce the example which we fully develop in section 4 to set out our method. Section 5 basically evokes some problems raised by the chosen approach.

II Some Remarks About Analogies Between Proofs

The following diagram shows how analogy is treated:

\[
\begin{align*}
\text{proof-schema1} & \quad \text{transformation rule} \quad \text{proof-schema2} \\
S & \quad \text{set of substitutions} \\
(\text{yield by 2nd order pattern matching algorithm}) \\
\text{proof1} & \quad \text{proposed-proofs} = \{\sigma(\text{proof-schema2}) | \sigma \in S\} \\
(\text{known proof})
\end{align*}
\]

In principle, analogies between proofs may be stated a posteriori in the metalanguage (using a (meta-)sentence expressing that a transformed term obtained from proof1 is itself a proof). This sentence can be proved in the metalanguage. But, in everyday mathematics analogies are used in a nonformal manner. When a mathematician wants to formally use a proof transformation, he does metareasoning and not analogical reasoning. Moreover analogy is intrinsically an uncertain way of reasoning, which, if used, must be checked.

The transformation rule heritates this intrinsic (and hazardous) uncertainty (it can denotes something which is not always true). In some way, the non-unicity of the solutions of the matching, as explained below, brings a part of this uncertainty.

Three questions arise naturally:

- Why natural deduction oriented?
- Why proof as term?
- Why second order pattern matching?

It is a well known fact that natural deduction is a good formalization of mathematical reasoning (see for ex. [Gentzen, 1969]) and the representation of proofs as terms reflects the abstract proof structure (see for ex.[Constable et al, 1985], [Constable et al, 1986], [deBruijn, 1980], [Miller and Felty, 1986]). We have thus adopted this paradigm in our approach.

Proof-representing terms are built from functional constant symbols denoting inference rules and first-order constants denoting axioms.

Having first-order variables in terms allows representation of partial proofs, which means proofs "containing" unproved lemmas (see [Gordon et al, 1979], [Milner, 1985]).

That is, these first-order variables range over proof terms. A further generalization will allow us to use variables to denote inference rules or composition of inference rules (considered as functions). This is obviously not possible if we restrict ourselves to first order terms, where function symbols are all constants.

III Technical Framework

We adopt the set of inference rules found in [Miller and Felty, 1986] which is a slightly modified version of Gentzen's LK system [Gentzen, 1969]. We list below only the ones we use in the following example.

\[
\begin{align*}
A, \Gamma \rightarrow \Theta \quad B, \Gamma \rightarrow \Theta & \quad \text{and1} \\
& \quad \text{A \land B, \Gamma \rightarrow \Theta} \\
\Gamma \rightarrow \Theta, A, B, \Delta \rightarrow \Lambda & \quad \text{imp1} \\
& \quad \Gamma \rightarrow \Theta, A \rightarrow B \quad \text{imp2} \\
[z, t]P, \Gamma \rightarrow \Theta & \quad \text{all1} \\
& \quad \Gamma \rightarrow \Theta, [z]tP \quad \text{some r} \\
A, \Gamma \rightarrow \Theta & \quad \text{thin1} \\
& \quad \Gamma \rightarrow \Theta \quad \text{axiom}
\end{align*}
\]

These inference rules considered as functional constants have polymorphic types. We write \( t : T \) to say that \( t \) is a well-formed proof term and \( T \) is a ground type (i.e. a sequent) which is an instance of the principal type of \( t \). Actually, it does not say more than: \( t \) is a proof of \( T \). Well-formedness and type inferencing on terms in polymorphic signatures are quite difficult problems and we shall not discuss them here. In the following we shall assume available a decision procedure for the correctness of such an expression \( t : T \).

In the following, we use a second order pattern matching algorithm from [Huet and Lang, 1978]. This al-
The proposed method on a detailed example

Let us now try to prove the following sequent:

\[ \text{seq2: } \vdash (p(a) \lor q(b)) \land \forall z(p(x) \Rightarrow q(z)) \land \forall z(q(z) \Rightarrow r(x)) \Rightarrow \exists x(r(x)) \]

Of course, one can prove it without any knowledge about the proof of seq1, but it may be easier to use some information carried by proof1. Moreover, we think that the human reader, having read and understood the proof of seq1, cannot try to prove seq2 without using proof1, at least unconsciously.

The usual way to use a proof is to have it as a subterm of the proof we are looking for. In our example, we can see that it is certainly possible here also using the lemma:

\[ \vdash \forall z(p(x) \Rightarrow q(z)) \land \forall z(q(z) \Rightarrow r(x)) \Rightarrow \forall z(p(x) \Rightarrow r(x)) \]

But this actually implies some metalevel reasoning (one can replace a subformula by an equivalent one, etc.), and all the process of proving metatheorems and using them is a quite difficult and long task. We shall not always want or be able to find and prove general results during the mathematical work.

Our goal here is to draw closer to informal remarks we can make after a quick analysis of proof1.

1. The last three rules are \( \text{imp}_r(\text{and}_L(\text{or}_L \ldots )) \).
   They are used to connect and transform the \( p(a) \) case and the \( q(b) \) case into the right sequent seq1.
   The only change to prove seq2 will be to add an \( \text{and}_L \) rule to "connect" the extra hypothesis \( \forall z(q(z) \Rightarrow r(x)) \).

2. On the right hand side of the tree, there is a "quick" reasoning on \( q(b) \), which we call \( g \), followed by an application of the thinning. To prove seq2, we shall have to add one thinning.

3. On the left hand side of the tree, we can find the same quick reasoning \( g \), but this time applied to \( q(a) \). Then follows something, say \( k \), to get the \( p(a) \) case.

At this point of analysis of proof1, or, we could better say, at this level of analogy between proof1 and proof2, we can write a transformation rule:

\[ f(\text{or}_L(k(g(q(a))))), i(\text{thin}_L(q(g(b)))))) \longrightarrow f(\text{and}_L(\text{or}_L(k'(g(r(a)))), i(\text{thin}_L(q(g(r(b)))))))) \]

where \( f, g, i, k \) and \( k' \) are second order variables with types:

\[ f : \Theta \rightarrow \Theta' \]
\[ g : \Theta' \rightarrow \Theta'' \]
\[ k, k' : \Theta'' \rightarrow \Theta \]

Of course, one can be more precise in giving a type to these variables, depending on the polymorphic possibilities. As above, we do not discuss this topic. Moreover Huet's second order pattern matching runs on a slightly restricted second order \( \lambda \)-calculus with simple types (sorts) (see [Huet and Lang, 1978] and also [Bundy, 1983]). We thus cannot (for the moment) use the polymorphic type discipline in the pattern matching, the only consequence of which is to bring more unifiers, the extra ones being useless.

There are some remarks to do concerning this transformation rule:

- We have introduced the variables \( f \) and \( i \) because we are not only interested in the analogy between proof1 and proof2 (the proof of seq2 we are looking for), but we have in mind a more general analogy.

- The variable \( k' \) only appears in the right hand side of the rule, and thus it cannot be instantiated by any unifier resulting from the matching with the left hand side. Therefore, this transformation rule does not bring proof terms, but proof schemas. The free variables appearing in them are to be instantiated by a theorem prover (the type of these instantiations is known given the sequent to be proved by the schema. Instantiating a first order variable is to prove a lemma, instantiating a second order variable is to find a deduction).
The pattern matching applied to proof1 with the left hand side of the rule gives a set of 14 unifiers (we have implemented Huet’s matching algorithm in Common-Lisp running on SUN), and thus we obtain 14 terms by applying these unifiers to the right hand side. We do not list them all, as most of them are to be deleted by the type inferencing process (given seq2).

In this example, the only remaining term is:

\[ \text{imp_r}(\text{and_l}(\text{and_l}(\text{or_l}(\text{imp}(\text{axiom}(r(a)))), \text{thn_l}(\text{thn_l}(\text{some_r}(\text{axiom}(r(b)))))))) \]

and \( k' \) must be instantiated with the type:

\[ r(a) \rightarrow \exists x r(z) \]

\[ p(a), \forall x (p(x) \rightarrow q(x)), \forall x (q(x) \rightarrow r(z)) \rightarrow \exists x r(z) \]

This is of course possible using the unifier:

\[ \lambda x. \text{all_l}(\text{imp_l}(\text{axiom}(p(a))), \text{all_l}(\text{imp_l}(\text{axiom}(q(a)), x)))) \]

where \( x \) is a first order variable of type \( r(a) \rightarrow \exists q(x) \).

This gives proof2 of type seq2. Furthermore, we can apply the transformation rule to proof2 to find a proof of seq3, that is:

\[ \rightarrow ((p(a) \lor q(b)) \land \forall z (p(z) \Rightarrow q(z)) \land \\
\forall x (q(x) \Rightarrow r(z)) \land \forall x (r(z) \Rightarrow s(x))) \Rightarrow \exists x \forall y \exists x \]

For that purpose, the rule must be slightly modified: replacing \( q \) by \( r \) and \( r \) by \( s \). The result of the pattern matching is a set of 21 unifiers, and at the end we obtain the term

\[ \text{imp_r}(\text{and_l}(\text{and_l}(\text{and_l}(\text{or_l}(\text{imp}(\text{axiom}(s(a))))), \text{thn_l}(\text{thn_l}(\text{thn_l}(\text{some_l}(\text{axiom}(s(b)))))))))) \]

and to get proof3, \( k' \) is replaced by

\[ \lambda x. \text{all_l}(\text{imp_l}(\text{axiom}(p(a)), \text{all_l}(\text{imp_l}(\text{axiom}(q(a)), \text{all_l}(\text{imp_l}(\text{axiom}(r(a)), x)))))) \]

This is the more general analogy we were talking about.

Now that we feel more comfortable with the problem of expressing analogies with transformation rules, we may refine the analogy between proof1 and proof2 to get a better transformation rule, i.e. such that there will be no free variables to be instantiated by a theorem prover.

The troublesome fact in the previous analogy lies in the third point, where we didn’t try to look into the “something” to get the \( p(a) \) case. But with a further analysis of the proof, we can see how it works. This “something” is built from a repetition of “something else”, say \( h \), on \( p(a) \), then on \( q(a) \) and so on...

There is still some fuzzyness remaining in that description. If we want, we surely could be more precise, and so on until we would find (alone...) the searched proof. We will rather let the computer do these last steps by its own, and thus we don’t mind that “something else”. Let us write the transformation rule expressing this level of analogy:

\[ f(\text{or_l}(\text{k(h(p(a), q(a)))), \text{i(thn_l}(\text{j(g(b))))})) \rightarrow \]

\[ f(\text{and_l}(\text{or_l}(\text{k(h(p(a), h(q(a), g(r(a)))), \text{i(thn_l}(\text{j(thn_l}(\text{j(g(r(b))))))))})) \]

In this rule, all the free variables in the right hand side are free in the left hand side. That is what we were looking for, but does it work, does it actually build the searched proof?

The only way to know is to try! The pattern matching with proof1 computes 64 unifiers. In the 64 terms then obtained we can find proof2. The corresponding unifier is:

\[
((g \text{ lambda } (x) (\text{some_r} (\text{axiom} x)))
(i \text{ lambda } (x) x)
(h \text{ lambda } (x y) (\text{all_l} (\text{imp_l} (\text{axiom} x) y)))
(k \text{ lambda } (x) x)
(f \text{ lambda } (x) (\text{imp_r} (\text{and_l} x)))
\]

Therefore, this analogy is correct to get a proof of seq2 from proof1. Moreover, we can say it is complete as it doesn't leave anything to prove. Only proof checking (type inferencing) is needed here.

As in the previous analogy, this one can be used to solve some other problems, at least the demonstration of seq3. The pattern matching with proof2 using the rule where we have replaced \( p \), \( q \), and \( r \) by \( q \), \( r \), and \( s \) respectively, brings 148 unifiers, among which we find the right one to get proof3:

\[
((g \text{ lambda } (x) (\text{some_r} (\text{axiom} x)))
(h \text{ lambda } (x y) (\text{all_l} (\text{imp_l} (\text{axiom} x) y)))
(i \text{ lambda } (x) (\text{thn_l} x))
(k \text{ lambda } (x) (\text{all_l} (\text{imp_l} (\text{axiom} (p a))) x))
(f \text{ lambda } (x) (\text{imp_r} (\text{and_l} (\text{and_l} x))))
\]

We now set out in an algorithmic way the proposed method:

1. The user already has \( \vdash \text{proof1} : \text{thm1} \) and he has a formula (or sequent) \( \text{thm2} \) he wants to prove, which he thinks is a problem analogous to the solved one.

2. He writes a (or uses an already written) transformation rule \( \text{proof_sch1} \rightarrow \text{proof_sch2} \) containing first or second order variables.

3. The matching is done between \( \text{proof_sch1} \) and \( \text{proof1} \), computing a finite set \( S \) of unifiers.

4. The corresponding instances of \( \text{proof_sch1} \) and \( \text{proof1} \) are computed. Let \( T = \{\sigma(\text{proof_sch2})|\sigma \in S\} \).

5. The proof checking is attempted on every element of \( T \). We then get \( T' = \{t | t \in T \land \vdash \text{thm2}\} \).

6. If the terms in \( T' \) have free variables, a theorem prover tries to instantiate them all in every \( t \) in \( T' \). We then have \( T'' = \{\sigma t | t \in T' \land \vdash \sigma t : \text{thm2}\} \) (equal to \( T' \) if there is no free variables in \( T' \)).

7. If \( T'' \) is empty, the analogy fails. Otherwise it succeeds and we can choose for example the shortest proof in \( T'' \) if there are several.

At any step from 3 to 0 a failure test (on emptiness of the computed set) can be added.
V Conclusion And Future Work

We have presented a method which we consider to be a first step towards a partial solution of the problem: "How to formulize a notion as powerful and frequently employed in human mathematical reasoning as proof analogy?"

Many problems strongly related to the principal subject of the present work have not been treated here. We are now working on them in order to have a deeper grasp of the ideas evoked in this paper. The problems are essentially those mentioned in 6,7 in the Introduction and they are:

- Some possible modifications of Huet's matching algorithm:
  - Try to take into account full types (to make types represent sequents). It will drastically diminish the number of unifiers, thus increasing the efficiency of the algorithm.
  - Maybe eliminate from the result the constant functions (i.e. \( \lambda x.t \), where \( x \) does not appear in \( t \)) which are not, a priori, useful in analogy. We do not necessarily need a complete set of unifiers.
  - The matching algorithm works on \( \lambda \)-terms. We only use a subset of this.
- Can we have more powerful expressing facilities in the language to write transformation rules?
- Is it possible to help the user to improve a firstly wrong or not quite interesting transformation rule, exploiting failures of the matching algorithm? More generally, is it possible to automatically build and use these rules?
- Is it possible to incorporate the kind of analogy presented in this paper to help (and hopefully guide) the "transformation tactics" presented in works as those of Constable et al. ([Constable et al, 1985], [Constable et al, 1986])?

Acknowledgments

We thank Ph. Schnoebelen for useful comments on an earlier draft of this paper.

References


