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— proposition, namely the one to be proved. This transformation is expressed by means of second order pattern matching (this may be seen as a generalisation 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.

I Introduction

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* "analogies". 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.

Partial support for this work was provided by the Centre National de la Recherche Scientifique (PRC Intelligence Artificielle)

- 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

> ≪ one can imagine writing very general "transformation tactics" (for details see [Constable *et al*, 1985]) to construct proofs by analogy to existing proofs ≫

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:

$$\begin{tikzcd} proof\_schema1 \arrow[r, "\text{transformation rule}"] & proof\_schema2 \arrow[d] \\ \underset{\text{(known proof)}}{proof1} \arrow[u, "\begin{array}{l}S\text{:set of substitutions}\\ \text{(yield by } 2^{nd}\\ \text{order pattern matching}\\ \text{algorithm)}\end{array}"'] & proposed\_proofs = \{\sigma(proof\_schema2) | \sigma \in S\} \end{tikzcd}$$

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.

$$\frac{A,\Gamma \to \Theta \quad B,\Gamma \to \Theta}{A \vee B,\Gamma \to \Theta}or\_l \qquad \frac{A,B,\Gamma \to \Theta}{A \wedge B,\Gamma \to \Theta}and\_l$$

$$\frac{\Gamma \to \Theta, A \quad B,\Delta \to \Lambda}{A \Rightarrow B,\Gamma,\Delta \to \Theta,\Lambda}imp\_l \qquad \frac{A,\Gamma \to \Theta, B}{\Gamma \to \Theta, A \Rightarrow B}imp\_r$$

$$\frac{[x|t]P,\Gamma \to \Theta}{\forall xP,\Gamma \to \Theta}all\_l \qquad \frac{\Gamma \to \Theta,[x|t]P}{\Gamma \to \Theta,\exists xP}some\_r$$

$$\frac{\Gamma \to \Theta}{A,\Gamma \to \Theta}thin\_l \qquad \frac{}{\Gamma \to \Gamma}axiom$$

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 algorithm receives on input a subset of second order λ-calculus which is enough here, and computes a complete and minimal set of unifiers. See [Huet and Lang, 1978] for details.

We are now going to develop an example (from [Miller and Felty, 1986]) to set out the different steps of the proposed method. The starting point is a proof of the sequent $seq1$: $\to (p(a) \vee q(b)) \wedge \forall x(p(x) \Rightarrow q(x)) \Rightarrow \exists x q(x)$

The proof, which we shall call $proof1$, is:

$$\dfrac{\dfrac{\dfrac{\dfrac{p(a) \to p(a) \quad \dfrac{q(a) \to q(a)}{q(a) \to \exists x q(x)}\,some\_r}{p(a), p(a) \Rightarrow q(a) \to \exists x q(x)}\,imp\_l}{p(a), \forall x(p(x) \Rightarrow q(x)) \to \exists x q(x)}\,all\_l \quad \dfrac{\dfrac{q(b) \to q(b)}{q(b) \to \exists x q(x)}\,some\_r}{q(b), \forall x(p(x) \Rightarrow q(x)) \to \exists x q(x)}\,thin\_l}{\dfrac{p(a) \vee q(b), \forall x(p(x) \Rightarrow q(x)) \to \exists x q(x)}{(p(a) \vee q(b)) \wedge \forall x(p(x) \Rightarrow q(x)) \to \exists x q(x)}\,and\_l}\,or\_l}{\to (p(a) \vee q(b)) \wedge \forall x(p(x) \Rightarrow q(x)) \Rightarrow \exists x q(x)}\,imp\_r$$

$proof1$ is represented by the term:

$$imp\_r(and\_l(or\_l(all\_l(imp\_l(axiom(p(a)), some\_r(axiom(q(a))))), thin\_l(some\_r(axiom(q(b)))))))$$

We thus have $\vdash proof1 : seq1$.

IV The Proposed Method On A Detailed Example

Let us now try to prove the following sequent:

$seq2$: $\to (p(a) \vee r(b)) \wedge \forall x(p(x) \Rightarrow q(x)) \wedge \forall x(q(x) \Rightarrow r(x)) \Rightarrow \exists x r(x)$

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: $\to \forall x(p(x) \Rightarrow q(x)) \wedge \forall x(q(x) \Rightarrow r(x)) \Rightarrow \forall x(p(x) \Rightarrow r(x))$

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 $imp\_r(and\_l(or\_l\ldots)))$. 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 and_l rule to "connect" the extra hypothesis $\forall x(q(x) \Rightarrow r(x))$.
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:

$f(or\_l(k(g(q(a))), i(thin\_l(g(q(b)))))) \longrightarrow$
$f(and\_l(or\_l(k'(g(r(a))), i(thin\_l(thin\_l(g(r(b))))))))$

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

$$f : \frac{\Theta \to \Theta'}{\to \Theta''} \qquad g : \frac{\Theta}{\Theta' \to \Theta''}$$

$$k, k' : \frac{\Theta' \to \Theta''}{p(a), \Gamma \to \Theta''} \qquad i : \frac{\Gamma \to \Theta}{\Gamma, \Gamma' \to \Theta}$$

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 λ-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 instanciated 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 instanciated by a theorem prover (the type of these instanciations is known given the sequent to be proved by the schema. Instanciating a first order variable is to prove a lemma, instanciating 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:

$$imp\_r(and\_l(and\_l(or\_l(k'(some\_r(axiom(r(a)))), thin\_l(thin\_l(some\_r(axiom(r(b))))))))$$

and k' must be instanciated with the type:

$$\frac{r(a) \rightarrow \exists x r(x)}{p(a), \forall x(p(x) \Rightarrow q(x)), \forall x(q(x) \Rightarrow r(x)) \rightarrow \exists x r(x)}$$

This is of course possible using the unifier :

$$< k'; \lambda x.all\_l(imp\_l(axiom(p(a)), all\_l(imp\_l(axiom(q(a)), x)))) >$$

where x is a first order variable of type $r(a) \rightarrow \exists x q(x)$. This gives $proof2$ of type $seq2$. Furthermore, we can apply the transformation rule to $proof2$ to find a proof of $seq3$, that is:

$$\rightarrow [(p(a) \vee s(b)) \wedge \forall x(p(x) \Rightarrow q(x)) \wedge \forall x(q(x) \Rightarrow r(x)) \wedge \forall x(r(x) \Rightarrow s(x))] \Rightarrow \exists x s(x)$$

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

$$imp\_r(and\_l(and\_l(and\_l(or\_l(k'(some\_r(axiom(s(a)))), thin\_l(thin\_l(thin\_l(some\_r(axiom(s(b)))))))))))$$

and to get $proof3$, k' is replaced by

$$\lambda x.all\_l(imp\_l(axiom(p(a)), all\_l(imp\_l(axiom(q(a)), all\_l(imp\_l(axiom(r(a)), x))))))$$

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 instanciated 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:

$$f(or\_l(k(h(p(a), g(q(a)))), i(thin\_l(g(q(b)))))) \longrightarrow f(and\_l(or\_l(k(h(p(a), h(q(a), g(r(a))))), i(thin\_l(thin\_l(g(r(b))))))))$$

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 lambda (x) (some_r (axiom x)))
 (i lambda (x) x)
 (h lambda (x y) (all_l (imp_l (axiom x) y)))
 (k lambda (x) x)
 (f lambda (x) (imp_r (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 lambda (x) (some_r (axiom x)))
 (h lambda (x y) (all_l (imp_l (axiom x) y)))
 (i lambda (x) (thin_l x))
 (k lambda (x) (all_l (imp_l (axiom (p a)) x)))
 (f lambda (x) (imp_r (and_l (and_l x)))))
```

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

1. the user already has $\vdash proof1 : thm1$ and he has a formula (or sequent) $thm2$ 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 $proof\_schema1 \longrightarrow proof\_schema2$ containing first or second order variables.
3. the matching is done between $proof\_schema1$ and $proof1$, computing a finite set S of unifiers.
4. the corresponding instances of $proof\_schema2$ are computed. let $T = \{\sigma(proof\_schema2) | \sigma \in S\}$.
5. The proof checking is attempted on every element of T. We then get $T' = \{t \in T | \vdash t : thm2\}$.
6. if the terms in T' have free variables, a theorem prover tries to instanciate them all in every t in T'. We then have $T'' = \{\sigma t | t \in T' \wedge \sigma t \text{ is ground} \wedge \vdash \sigma t : thm2\}$ (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 6 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 formalize 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 λ-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

[Boyer and Moore, 1984] Robert S. Boyer and J. Strother Moore. Proof Checking, Theorem Proving and Program Verification. In *Automated Theorem Proving: After 25 Years*, pages 119-132. Contemporary Mathematics Vol. 29, American Mathematical Society, 1984.

[Bundy, 1983] Alan Bundy. *The computer modelling of Mathematical Reasoning.* Academic Press, 1983.

[Constable *et al.*, 1985] Robert L. Constable, Todd B. Knoblock, and Joseph L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning* 1:285-326, 1985.

[Constable *et al.*, 1986] Robert L. Constable *et al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[Davis and Schwartz, 1979] Martin Davis and Jacob T. Schwartz. Metamathematical extensibility for theorem verifiers and proof-checkers. *Comp. & Maths. with Appls.*, 5:217-230, 1979.

[deBruijn, 1980] J. G. deBruijn. A Survey of the project AUTOMATH. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* , pages 579-606. Academic Press, 1980.

[Gentzen, 1969] Gerhard Gentzen. Investigations into Logical Deduction. in *The collected papers of Gerhard Gentzen*, pages 68-131. North-Holland, 1969.

[Gordon *et al.*, 1979] Michael J. Gordon, Arthur J. Milner and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation.* Lecture Notes in Computer Science 78, Springer-Verlag, 1979.

[Huet and Lang, 1978] Gerard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns.*Acta Informatica* 11:31-55, 1978.

[Huet, 1986] Gerard Huet. *Deduction and Computation.* Rapport de Recherche INRIA No. 513, April 1986.

[Kling, 1971] Robert E. Kling. A paradigm for reasoning by analogy. *Artificial Intelligence* 2:147-178, 1971.

[Knoblock and Constable, 1986] Todd B. Knoblock and Robert L. Constable. *Formalized Metareasoning in type theory.* TR 86-742, Cornell University, March 1986.

[Miller and Felty, 1986] Dale Miller and Amy Felty. An integration of resolution and natural deduction theorem proving. In *Proceedings AAAI-86*, pages 198-202. Philadelphia, Pennsylvania, American Association for Artificial Intelligence, August 1986.

[Milner, 1985] Robin Milner. The use of machines to assist in rigorous proof. in *Mathematical Logic and Programming Languages*, pages 77-88, Prentice-Hall 1985.

[Plaisted, 1981] David A. Plaisted. Theorem proving with abstraction. *Artificial Intelligence* 16:47-108, 1981.

[Takeuti, 1975] Gaisi Takeuti. *Proof Theory.* North-Holland 1975.