How to disambiguate unification in minimal proof tools

Philippe Ladagnous
IRIT, UPS Toulouse, 118 Route de Narbonne,
F-31062 TOULOUSE Cedex, FRANCE
Phone number: +33.5.61.55.63.25

Abstract: The aim of the mechanism described in this paper is to ease the verification of UNITY programs. The basic idea is that in many theorem provers such as Deva[12], the user spends too much time describing what he wants to do. This waste of time may come from the search for the correct rule to apply on the one hand and from the several ways of applying it on the other. So, we provide in our "Devaur" tool a functionality to overcome these difficulties through the analysis of user mouse manipulations. But whereas other Devaur-like tools have statically defined such interaction interpretation, we provide a language allowing the developers of new Deva libraries to define their own ones. When several theories are involved, all these interpretations coexist and may generate several different rules that could be applied. So, at each proof step, several rules may be proposed, but since they are focused on the user selection, they are always pertinent. Moreover, due to our language, they may be suggested through natural sentences, and so, very easy to select or discard. Furthermore, since the difficulties this mechanism can overcome are commonly met in minimal theorem provers, we think it could be an inspiration for some of them.

Keywords: UNITY, theorem prover, higher order unification, graphical user interface (GUI), mouse interactions interpretations

1 Introduction

Writing correct parallel programs is a very difficult task and consequently, we would like to formally check such developments. In order to do that, we need a formalism to express specifications and programs, and a tool to ensure that such programs respect their specification. The genericity of the Deva language [12] in terms of development methods together with the fact that an interpreter of this language has been developed in our team [7] led us to choose it as the base support for our developments.

On an other hand, researches already conducted in this domain by an other IRIT team [4] led us to choose UNITY [3] as the formalism to express parallelism. After having developed Deva libraries for UNITY, we unfortunately noticed that, due to the minimality of the proof system, writing development proofs is very long and tedious and so unrealistic for most of them.

From our experience, we notice that in fact, even if the user has in mind how to conduct the proof, he often wastes time searching the right rule to do it. Indeed, even if he may get from the tool the list of applicable rules, searching the required one among a possibly very large set can be quite long. Moreover, the higher order of the Deva unification process can lead to indeterministic rule application[9], and again, some time is spent making it deterministic (before application) or searching the correct variant (after application).

The underlying idea of the mechanism presented here is, as in many tools of this kind, to get from the user a complementary information in order to both reduce the number of suggested rules and generate a determinised version of each rule.

The main difference between our mechanism and those traditionally implemented is its genericity. Indeed, a system like CtCoq[2, 1] implements statically defined interpretations of user interactions. For example, a "drag and drop" operation from a binary operator towards one of its

arguments is reduced into its second argument if the first argument is a neutral element for the operator. Even if the developer is able to define new operators and their associated neutral elements, the general behaviour of the interface will always remains the same. In a similar way, the "proof by pointing" mechanism [2] is statically defined and, when performed in a same context, user interactions are always interpreted in the same way. In Devaur, we take a different approach : a theory developer (i.e. a person who implements a new Deva library in order to deal with a new theory) is able to precisely specify how and when each Deva rule of this new theory should be applied.

We present in the next section an example of the problems we want to deal with. A concrete solution is given in section 3. A way of introducing humanity in these new results is then presented in section 4. The fifth section introduces higher order tactics to allow larger proof steps, then a conclusion and prospects are given in section 6.

2 Indeterministic Rules

We take a simple example: the induction principle on naturals numbers. It is defined in the following way in our libraries :

```
induc : [P?[N |- B]; P(0); [n:N; P(n) |- P(s n)] |- [n:N |- P(n)]]
```

Here is an informal description of this rule : for all predicate P whose domain is N, if we provide a proof of $P(0)$ and a proof of $\forall n.P(n) \Rightarrow P(s(n))$, then we get a proof of $\forall n.P(n)$.
If we attempt to apply in a backward chaining way the *induc* rule on a goal such like $x*(y+z)=x*y+x*z$, Deva tries to unify the goal and the rule conclusion : the unification process then returns several different possible applications; in fact, it returns one variant for each natural subterm and one for each combination of syntactically equal natural subterms (and not only variables) of the goal. Unfortunately, whereas only three of these variants have a mathematical sense (the induction on *all* occurrences of x, y or z), there is no way to specify it in Deva. Moreover, concerning the request of applicable rules, since the *induc* rule conclusion is $P(n)$, it can always be applied when the goal is a boolean expression which contains a natural subterm; since it is a very common situation, this rule will be very often suggested.
In brief, the *induc* rule is very often proposed[1] and its application is generally indeterministic : this rule is really a good example of bad Deva rule.

Let's now suppose we get a complementary information from the user : a subterm (or the whole goal itself) of the current goal; then, it is quite logical to propose the *induc* rule only when the user selects a natural variable. Moreover, the same information can be used to determinise the rule: the tool knows on which variable the user wants to induct. Unfortunately, we will see later that this user selection interpretation is really bound to this particular rule and that only the theory developer is able to give a pertinent sense to user selection, this is the purpose of the language of "tactics".

3 Making the induction rule deterministic

Let us introduce a part of this new language through the *induc* tactic :

```
tactic induc  :: (((type-of(goal)  equal B)  and (type-of(sel) equal N))
                    and (sel match ident(i)))
              :   return (induc(P:=[newvar:N |- goal[newvar // sel][2]])
                       ->  goal[0 // sel]
```

[1] Since the aim is to propose only a few rules at each proof step, it is not really an advantage.
[2] The notation "t1[t2//t3]" stands for t1 where all occurrences of t2 are replaced by t3.

```
      , [h: goal[inducv // sel] |-  goal[s(inducv) // sel]]
    --> Induction on sel)
```

A tactic is composed of three parts; in the first one stands the name "induc" of the tactic. The second part (between "::" and ":") is a conditional part allowing us to describe in which cases the tactic may be used. This part is in fact a boolean expression. In the example, we impose that : (1) the current goal (denoted by the "goal" keyword) is boolean (due to the boolean nature of the *induc* rule conclusion (2) the user selection (denoted by the "sel" keyword) is of type **N** (of course : this is the induction on naturals!) (3) the selection must be a Deva identifier (induction can be only applied on variables).

The third part of the tactic contains the instruction. In this simple case, it is reduced to a *return* instruction which essentially returns the Deva rule to send if the tactic is applied. Two optional arguments may be added, the first one is the new possible subgoals the rule application will introduce and the second is described in the next section.

If applied, this tactic sends the *induc* rule with an explicit value for its parameter P. This value is defined as an abstraction whose conclusion is the current goal where all subterms syntactically equal to the user selection are substituted by the abstraction parameter; the unification process of $P(n)$ with the goal then returns only one result : the application is deterministic.

For example, if the user clicks on y in $x*(y+z)=x*y+x*z$, then the unification process of the current goal with $induc(P := [newvar : N \vdash x * (newvar + z) = x * newvar + x * z])$ leads to only one result.

4 Introducing Humanity in Tactics results

Since the tactic mechanism can propose several Deva rule applications, we have to help the user choosing the right one; for that purpose, we provide two ways of presenting these rules : the first and most classical is to display for each tactic what (possible) subgoals will appear if it is applied. But even if tools like Devaur are rigorous and formal, when no confusion can be introduced, speaking in natural language is a chance to introduce some humanity in them. Thus, we provide the ability to present each tactic through natural sentences : this is the purpose of the optional *return*'s third parameter. Naturally, the tactic variables and keywords such as *goal*, *sel* or *type-of* may be used. For example, with the previous definition of the *induc* tactic and the selection of the variable y, the message "Induction on y" appears in place of the more traditional base and iteration cases.

5 UNITY properties and higher order Tactics

Recalling that the main purpose of our tool is to make the proofs of UNITY programs easier, we now introduce more complicated tactics intended to reduce the amount of user work during this process. In order to do that, we must present (unfortunately very succinctly) how we deal with UNITY programs and properties.

In Devaur[10, 11], as in many systems[8, 6, 5], the semantics of a UNITY program is described through the set of all its possible execution sequences, where a sequence is an ordered infinite list of states, each state attributing (or not) a value to each variable. For some reasons we do not describe here, we prefer to create a new sort for states (which is called "*st*") and to consider variables as functions from *st* onto their domain (integer or boolean) rather than the inverse (which is widely used). Here are the corresponding assertions we can find in Devaur :

st : sort	Declaration of a new sort for states
si := [st $\vdash$ I]	Abbreviations to make new integer or boolean state variable declara-
sb := [st $\vdash$ B]	tions easier.

We define state expressions as Deva texts inductively build starting from state variables constants and operators; for example, if A and B are of type "sb", A $\wedge*$ B is a state expression ($\wedge*$ is the conjunction of state booleans). We now define the polymorphic equality on state expressions and extend the commutative nature of addition onto state integers :

```
    =*          : [ s ? sort; [st ⊢ s] ; [st ⊢ s] ⊢ sb ]
 adde_commut    : [x,y ? si ⊢ x +* y =* y +* x]
```

Declaring an abstraction parameter through a question mark means that no explicit corresponding argument must be provided during the rule application and Deva will automatically synthesize the correct value. Thus, the "=*" operator takes only two arguments of type "[st $\vdash$ s]" (s can be of any non functional type) and returns a state boolean : the equality of two state expressions depends on the considered state. We now extend the substitutivity principle of equality over state equality :

```
subst_pg : [ s?sort ; x,y ? [ st ⊢ s ] ; P ? [ [ st ⊢ s ] ⊢ B ] ; x =* y ; P(y) |- P(x) ]
```

This rule states that for two state variables x and y of type s, and a predicate P on variables of this type, if we can prove the equality of x and y and that $P(y)$ holds, then we get a proof of $P(x)$.

This rule allows us to exchange two equal state expressions in a boolean property. In fact, it has been written in order to manipulate properties of the UNITY logic. Notice that the conclusion of this rule is an undefined predicate applied to an undefined state expression; here again, this rule conclusion can be unified with any boolean term that contains a state expression. It follows that this rule will be very often suggested and moreover, with a large set of variants. So, we define a tactic to overcome this fact :

```
tactic subst_pg_gen :: ((type-of(goal) equal B) and ([ty] match type-of(sel))
                                                 and (type-of([ty]) match [st|-sort]))
                     :   return(subst_pg(P:=[newvar:[ty] |- goal[newvar/sel][3]])
                                -> op(=*,newvar,sel) , goal[z/sel]
                               --> Substitution by a new free variable)
```

The right operand of the first "and" stores the type of the user selection in the unused "[ty]" variable; this value will be used during the creation of the predicate P. We then check that this value is correct in regard to the *subst_pg* rule. Indeed, this rule allows only state expressions substitutions. Finally, it returns the correct deterministic Deva rule. We notice that P is constructed starting from "goal" where only the selected subterm is replaced by the new variable and not all syntactically equal occurrences.

After its application, we get two new subgoals. Indeed, we must prove the previous goal in which the substitution has been performed and the equality between the exchanged subterms. Practically, the substitution principle is very often used to exchange two subterms that can be proved equal in one step (i.e. by application of a unique rule that do not generate any new subgoal). In this case, it would be interesting to use this so called "trivial" rule directly in the tactic and hence to remove the second previously mentioned goal. In fact, when the user selects a subterm, we should search for "trivial" rules that can prove the selected subterm equal to an other one and then suggest the corresponding substitution. In this way, by using this "trivial" rule directly as an argument of the substitution rule, we get just one new subgoal : the previous goal on which the substitution has been applied.

```
tactic subst_pg :: ((type-of(goal) equal B) and ([ty] match type-of(sel))
                                            and (type-of(type-of(sel)) equal [st|-sort]))
```

[3] We denote by t1[t2/t3] the term t1 where the subterm t2 is replaced by t3.

```
:   foreach tactic (op(=*,[z],sel))
    do return(subst_pg(P:=[newvar:[ty] |- goal[newvar/sel]])
                -> goal[[z]/sel]
               --> Substitution by [z])
```

The role of the "foreach tactic" instruction is to search for trivial tactics that can prove a given goal, *op(=*,[z],sel)* in the example. Let's suppose T is such a tactic, meaning that T returns a trivial Deva rule t which proves *[z] =* sel*; moreover, it comes from the trivial nature of t that [z] will get a concrete value during this process.

Now, if we send to Deva the rule *subst_pg(P:=[newvar:[ty] ⊢goal[newvar/sel]]) (t)*, t will be considered as the text proving the $x =^* y$ hypothesis of the *subst_pg* Deva rule (due to the fact that the previous parameters of this rule are implicit). Since t proves *[z] =* sel*, *[z]* getting a concrete value, it follows that the x et y parameters of *subst_pg* get concrete values too. So, it comes that the *newvar* parameter of the P predicate gets also a concrete value.

Finally, applying *subst_pg(P:=[newvar:[ty] ⊢goal[newvar/sel]]) (t)* entirely defines values for all concerned variables and moreover, provides a proof for the $x =^* y$ hypothesis of the *subst_pg* rule. Then, only the proof obligation of *goal[[z]/sel]* remains, that is the previous goal where the selection has been replaced by a known and proved equal term.

Since the "foreach tactic" instruction performs this work for each trivial tactic, it returns the complete set of possible substitutions for the selected subterm and of course the associated composed Deva rules.

More generally, this instruction is used in tactics associated with non trivial Deva rules (i.e. that generate several subgoals if applied without any argument). For example, if a rule r is of the form *[h1:t1;...;hn:tn ⊢C]*, just applying r will generate n new subgoals; if we use the "foreach tactic" to search for trivial tactics able to prove *h1* and if we send to Deva the application of r on one of these rules, we then will get only *n-1* subgoals. This is especially interesting when n equals two, because we just get an only new goal in place of the previous one, one can say the application is transparent for the user.

Since UNITY properties are boolean predicates on state expressions, the *subst_pg* tactic can be applied on such properties. For example, if the current goal and selection are "$\boxed{x +^* y} =^* 0$ *unless* $x +^* y >^* 0$ *in CPT*", whereas asking the list of applicable rules returns many possibilities, the tactic mechanism just suggests the addition commutativity, the substitution by a new variable and two others rules. Moreover, they are proposed through natural sentences such like "Substitution by y +* x" or "Substitution by (x +* y) +* 0".

6 Conclusion and prospects

Term manipulations is really a hard problem in proof tools and mechanisms are very often provided in order to deal with them. Since Deva is not an exception to this rule, we provide an environment, named "Devaur", among whose functionalities stands a "tactic" mechanism. The purpose of this tool is to interpret user mouse interactions in order to determinise certain Deva rule applications. Since we find many kind of rules, and since many of them should be used differently, we prefer to provide a way for the theories developers to express these rules particularities through a new dedicated language.

Even if an intensive use of the mechanism, in particular when proving UNITY properties, led us to really appreciate its power, we nevertheless regret the wasting of computer resources. Indeed, since the tactic mechanism does not stand in Deva, a large part of data and algorithms can be found in both tools. By chance, since the UNITY formalism is not so large, the work on a Sparc classic Workstation with 72Mb of memory is still comfortable. With about ten higher order tactics and approximatively thirty first order tactics, it takes between one and two seconds to get the result

of the tactics search. This time is quite acceptable, in particular when we compare with the time spent to disambiguate "by hand" the unification process.

An advantage of having developed it over Deva is its possible reuse for some others tool provers. Moreover, due to the classical way of storing Deva texts as trees in Devaur, and since many tools use this data structure too, communication process between them should not be so difficult to implement. Furthermore, since our tactics mechanism seems to be quite general, we think it could be at least an inspiration for new such tools.

Finally, due to the latest Devaur evolutions such as automatic proofs on propositional calculus, Presburger arithmetic[13] and safety UNITY properties, the verification of UNITY developments becomes faster and faster. Moreover, even if this is probably not the best environment to develop UNITY programs, we like the idea of overcoming Deva gaps (in fact due to its minimality) by more complicated tools without loosing the simplicity (i.e. preserving the writing of proofs starting from a low level axiomatics).

Concerning our future aims, since the Deva interpreter we use does not provide any such functionality, we would like to extend our tool to provide a way of chaining tactics as in Coq or HOL. Indeed, since each tactic provides the list of future subgoals, we certainly can use them to explore the proof tree without Deva. Thus, we could provide a mechanism of "tacticals" able to search for and generate more complicated Deva rules in order to make larger proof steps.

References

[1] J. Bertot and Y. Bertot. Ctcoq: A system presentation. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 600–603. Springer-Verlag, 1996.

[2] Yves Bertot, Gilles Kahn, and Laurent Thery. Proof by pointing. *Lecture Notes in Computer Science*, 789:141–160, 1994.

[3] K.M. Chandy and J. Misra. *Parallel Program Design : A Foundation.* Addison-Wesley, 1988.

[4] M. Charpentier, M. Filali, P. Mauran, and G. Padiou. Tailoring UNITY to distributed program design. *Lecture Notes in Computer Science*, 1388:820–836, 1998.

[5] M. Charpentier, A. El Hadri, and G. Padiou. Preuve automatique dans un environnement unity. *T.S.I Technique et Science Informatique*, 15, january 1995.

[6] P. Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications - Application to UNITY* . PhD thesis, Université Catholique de Louvain, 1994.

[7] A. Coste. *Un Système pour la Validation des Développements de Programmes: Définition Formelle et Réalisation.* PhD thesis, Université Paul Sabatier - Toulouse, sep 1993.

[8] F. Andersen, K.D. Petersen, and J.S. Petterson. Program Verification using HOL-UNITY. In J.J. Joyce and C.-J.H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 1–16, Vancouver, Canada, August 1993. University of British Columbia, Springer-Verlag, published 1994.

[9] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[10] P. Ladagnous. Vers la vérification automatique de programmes UNITY. In Libert et Manneback Dekeyser, editor, *Renpar'7*, Mons, mai-juin 1995. Résumé et Poster.

[11] P. Ladagnous. Vers la vérification automatique de programmes UNITY(2). In *Actes du séminaire FAC'96*, IRIT, Toulouse, 1996.

[12] C. Lafontaine, M. Simons, and M. Weber. *The Generic Development Language Deva - Presentation and Case Studies*, volume 738 of *LNCS*. Springer-Verlag, 1993.

[13] M. Presburger. Über die vollständigkeit eines gewissen systems der arithmetic ganzer zahlen in welchem die addition als einzige operation hervortritt. In *Comptes-rendus du premier Congrès des Mathématiciens des Pays Slaves*, pages 192–201,395, Warsaw, 1929.