Head-tactics simplification

Yves Bertot

INRIA Sophia Antipolis

Abstract. Tactics are commands used to guide goal-directed proofs in interactive proof environments. This paper presents various possible simplifications on tactic expression and provides a justification for these simplifications, based on a precise description of the way tactics operate. In particular, this paper introduces a class of *head-oriented* tactics that are especially suited for simplification. Most of these simplifications have been developed in an simplifier coupled with a tactic generator based on mouse interaction.

1 Introduction

The work described in this paper deals with tools for interactive proof environments on the computer. The help that computers could bring in the development of proofs has been recognized for a long time. Two main domains have developed from this recognition. The first area is that of proof automation. In this domain, interactivity is considered harmful and researchers attempt to make the computer find complex proofs on its own. In this domain, one has to fight with incompleteness and complexity theorems, that express that there is no algorithm that will successfully detect true formulas, and, that even in domains where algorithms exist, these algorithms may often be useless because of the time they require to reach a solution. The second area is that of interactive proof development. In this domain, one still relies on a human operator to describe the main steps of the proof and the research goal is to make human operators as efficient as possible in their work to provide the proof steps.

Interactive proof development tools often contain two kinds of formal languages. The first one is used to represent the logical formulas, the second one is used to change the state of the proof environment, in order to progress towards a state where the validity of some formulas has been ascertained and mechanically checked. An important mode of operation is *goal-directed proof*, where one states a logical formula to prove and applies commands to reduce this command into simpler formulas. The commands that one applies in this mode are usually called *tactics*. The tactics form a sub-language of the command language of proof environments, with control and composition operators.

Recent studies in the user-interface of theorem proving tools [22] have shown that the characteristics of modern computer workstations could be used efficiently to improve the usage of proof environments. With features like *Proof-by-pointing* [2], it is possible to interpret simple tokens of information, like the position of the mouse with respect to a logical formula, to produce complex

in: *Algebraic Methodology and Software Technology (AMAST)*, Sydney, Australia, December, 1997.

tactics. In this paradigm, mouse interaction is considered a high-level language, compiled into the tactics language, considered as a low-level language. In different words, the proof-by-pointing algorithm is just a compiler, used thousands of times in one interactive session with the proof environment.

The results of the proof-by-pointing algorithm need to be simplified. The simplification is different from thue usual optimization phase of a compiler. Here the efficiency will vary little between the original expression and the simplified one. On the other hand, readability will be a significant criterion, since the aim of tools like proof-by-pointing is to construct commands that the user could have written by hand. Another important feature of this work is that the new expressions produced by the simplifier are semantically different from the original expressions, in ways that are justified by practical experience.

This simplifier is what we describe in this paper. In the second section, we give a short description of proof representation in type theory and use this framework to show how tactics operate, to present a few tactic composition constructs, and to introduce a class of tactics we call *head-oriented tactics*. In the third section, we describe several simplification rules on tactic expressions. The first simplification rule concerns only a composition construct. The next group of rules concern the tactics used to handle implication and *forall* quantification. The last rules concern the interaction of these tactics with head-oriented tactics. All these simplification rules are used in a user-interface we are developing for the Coq proof system [13].

1.1 Related work

In the study of proof systems, very little attention has been devoted to the language used to advance proof description, with one notable exception: the ML programming language [17, 19]. This functional language has now evolved into a general-purpose programming language, but it was originally intended as the command language for the Edinburgh LCF proof system. Its formal semantics have been thoroughly studied, as have the ways to compile and to execute it. The language is still evolving, and it is increasingly being used outside the proof community.

In the specific domain of tactics and tactic composition, precise descriptions of tactics are given in [7], [18], and [11], each time as an instanciation for their specific proof systems. A more abstract account of *tactic trees* was done in [12]. This account represents quite faithfully the notions of tactics that are used in this paper. The work described in [9] also contains an interesting study of tactics and their interaction with metavariables (called existential variables in this paper).

The presentation of proof systems described in this paper draws very much from the area of type theory, where proof checking is viewed as a generalization of program type-checking. This domain has been initiated by work of Martin-Löf [16] and has led to many implementations of proof systems [7, 13–15].

Work on proof-by-pointing is the initial incentive for this work on tactic simplification. Proof-by-pointing, where commands are generated from an interpretation of locations selected by the user, has been formally described in [2] and [4]. It has been implemented in several experiments of proof environments, based on a variety of proof systems: Isabelle [20], HOL [21], a theorem prover developped in λ-prolog [8], and Coq [13], all using a structure editor [22, 5] to facilitate the input of mouse location information. Although the initial algorithm is intended for "common" logics, extensions of this notion have also been studied for modal logic [10]. On the implementation side, the importance of structure editors for mouse interaction has also been studied and an experiment using the Lego proof system [14] and Emacs [6] have shown that this coupling could be lessened [3].

2 Domain of study

In this section we first describe how proof assistants can use the Curry-Howard paradigm to represent proofs and how this model can be used in uniform tactics.

2.1 The Curry-Howard isomorphism and head-oriented tactics

This section introduces the concepts of type theory and tactics to help understand the context of our work.

Types for proof assistants The Curry-Howard isomorphism relies on the strong similarity that there is between typing a program and checking a proof. It establishes a link between types and logical propositions on one side and programs and proofs on the other side. The most typical example revolves around the arrow, used in one case to represent function types and in the other case to represent implication. This correspondance is quite easy to understand intuitively: *a proof of $A \Rightarrow B$ maps any proof of A to a proof of B*, exactly in the same manner that a total function of type $A \rightarrow B$ maps any element of A to an element of B.

For an expressive formalism, it is necessary to go beyond the types regularly found in programming languages. For instance, a universally quantified formula $\forall x \in \mathcal{N}.P(x)$ also corresponds to a function type for functions that map any object n of type $\mathcal{N}$ to a proof of $P(n)$. One sees immediately that this function takes all its arguments in the same type $\mathcal{N}$ and maps them to *different* types: here the result type depends on the value of the argument, and not only on the argument type as one is accustomed to with polymorphic functional languages. It is possible to generalize the recursive data-structures found in functional languages to capture logical phenomena. For instance, a conjunction is very much akin to a pair type, while a disjunction is akin to a sum type.

Proving a theorem boils down to exhibiting an object whose type is the theorem's statement. Here, the two methods of forward proving and backward proving, as already known in the context of other logical systems like HOL or Isabelle are also available. When performing forward proving, you simply construct by hand a term with the right type. This is very often difficult because the type information that is requested by the proof system is very cumbersome to provide. Backward proving is also possible: first state the goal logical formula, then provide a function that might return an object of that type, if it were applied to arguments of the right type, then consider these new types as new subgoals and proceed recursively.

Backward proving proceeds by constructing terms with typed holes. At the beginning, there is only one hole and its type is the goal. When the user attempts to fill this hole in with a function whose arguments are undetermined new holes are created, presumably with simpler types, corresponding to simpler goals. This is top-down construction. This top-down construction is very well presented in Alf [15].

Beyond the idea of manipulating proof terms with typed holes, one might want to consider having holes in the types themselves, leading to situations where the exact statement of what needs to be proved will be decided as the proof goes on. The idea is quite appealing, especially for automatic proofs and when the proof of a statement depends on the existence of a term verifying some specific property. The holes are often referred to as *existential* variables and these holes are usually filled in using some form of unification (higher order unification in Isabelle for instance [20]). However, such existential variables usually end up occurring in several goals, that is, in the type of several holes and this breaks some form of regularity in the proving process: the proof of two distinct goals in a given incomplete proof term is not independent if these two goals share an existential variable. In the rest of this paper we will not consider proof term development with existential variables, so that the property that two subgoals can be proved independently holds.

Understandably, tactics-based proving is also available [18]. *Tactics* are functions that take a goal as argument and return a new list of subgoals and a validation, with the intended meaning that the validation makes it possible to construct a proof of the goal as soon as it is given a list of proofs for the subgoals. Here the validation simply needs to be an incomplete proof term, that will have the type of the goal as soon as one provides the subterms corresponding to its holes.

Tactics operation

Introduction and elimination of $\forall$ and $\Rightarrow$ Many theorems have a statement in a *functional* form, that is, a universally quantified statement of the form $\forall x : A.B$ or an hypothetical statement of the form $A \Rightarrow B$. The natural form of proofs for these statements is that of a function, which takes an element of the given type as argument (here A in both cases) and has a body where this element can be used to construct an object of type B. It is natural to have a local context attached to each goal and a tactic, named `Intro`, that takes a goal of the form given above and produces a new goal with the statement B and a new declaration $x : A$ (meaning we have an object x of type A) in the local context. For the proof term construction, this corresponds to expressing that the proof will have the form of a lambda abstraction $\lambda x : A.M$ where M is the new hole.

In this framework, introducing a *cut* in the proof simply corresponds to introducing a β-redex. There are three possible forms of tactics that make this kind of manipulation possible:

1. There is a `Cut` tactic. When applied to a goal B, the tactic `Cut A` introduces two new goals $A \Rightarrow B$ and A. The proof term constructed has the form $M(N)$, where M must have type $A \Rightarrow B$ and N must have type A.
2. There is a `Generalize` tactic. When applied to a theorem `thm` of type A for a goal B, it introduces a goal of the form $A \Rightarrow B$. The proof term constructed has the form $M(\texttt{thm})$. This may look useless, but it will be used to make instances of universally quantified theorems appear (the operation named $\forall$-elimination) as follows: if the goal is C and if `thm` has the statement $\forall x : A.B(x)$ and if a has the type A, then `Generalize thm(`a`)` produces a goal of the form $B(a) \Rightarrow C$.
3. There is an `LApply` tactic. When applied to a theorem `thm` of type $A \Rightarrow B$ for a goal C, it produces the two goals $B \Rightarrow C$ and A. The proof term constructed has the form $M(\texttt{thm}(N))$ where M must have type $B \Rightarrow C$ goal and N must have type A.

In some respect, the `Cut` tactic and the `Generalize` and `LApply` tactics do not live in the same world, as they can be distinguished by the kind of arguments they take. The argument of the `Cut` tactic is a logical formula (a type) while the argument of the two other tactics is a proof term. From a practical point of view this difference is important: logical formulae tend to grow large while proof terms will more often consist in a single identifier (a theorem name) applied to a limited number of arguments.

Tacticals Tactics can be composed in several ways, for instance with repetitive statements, or conditionally, depending on the failure of previous tactics. In this section, we will only refer to constructions for sequence compositions.

The first sequencing construction corresponds to the `THEN` tactic of LCF-style provers. In our presentation it will be noted $\texttt{tac}_1\texttt{;}\ \texttt{tac}_2$. The behavior of this construction is to apply $\texttt{tac}_1$ on the goal at hand and then apply $\texttt{tac}_2$ on all the goals produced by $\texttt{tac}_1$. This construction is very supple with respect to the number of produced goals, but the user has to be sure that $\texttt{tac}_2$ will operate successfully on all of them.

The second sequencing construction corresponds to the `THENL` tactic of LCF-style provers. In our presentation it will be noted $\texttt{tac;[tac}_1\texttt{; ...;tac}_n\texttt{]}$. The behavior of this construction is to apply `tac` on the goal at hand, to verify that it produces n goals $G_1, \ldots, G_n$ and to apply $\texttt{tac}_1, \ldots, \texttt{tac}_n$ respectively to these goals. This construction is more rigid, as it expects a specific number of new subgoals, but the user can apply different tactics for different subgoals.

Paradoxically, a useful tactic that is complementary to the `THENL` tactical is a tactic that does nothing. When several goals are produced, it is important to be able to express that one will do something specifically on one of the subgoals, and nothing on the others. We shall name `Idtac` a tactic that succeeds on any goal and produces exactly the same goal.

Head oriented tactics As we already mentionned, theorems whose statement is an implication have a functional form. After applying one such theorem to the right number of arguments, one reaches a derived theorem whose statement is a plain subterm of the initial statement. In terms of proof term construction, this corresponds to filling a hole of type B with a theorem of type $A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow B$ applied to the n holes $A_1, \ldots, A_n$. A head-oriented tactic is a tactic that incorporates directly this behavior in its mode of operation.

Definition *A tactic* `tac` *is* head-oriented *if for any two theorems* $\mathtt{thm}_1$ *and* $\mathtt{thm}_2$ *with types* B *and* $A \Rightarrow B$ *respectively,* `tac` $\mathtt{thm}_1$ *and* `tac` $\mathtt{thm}_2$ *can be applied successfully whenever the other one can, and if* $G_1, \ldots, G_n$ *are the goals generated by the tactic* `tac` $\mathtt{thm}_1$ *then* $A, G_1, \ldots, G_n$ *are, in this order, the goals generated by* `tac` $\mathtt{thm}_2$.

Formally, the *head* and *depth* of a logical formula can be recursively described as follows:

1. The head of $\forall x : A.B$ is the head of B, the depth of $\forall x : A.B$ is the depth of B plus one,
2. the head of $A \Rightarrow B$ is the head of B, the depth of $A \Rightarrow B$ is the depth of B plus one,
3. if A does not fit in the previous patterns, the head of A is A, its depth is 0.

The head-oriented behavior can be extended to cope with *dependent* functional types (that is, universal quantification). The first means of extension appears when there is a way to know which specific instance of the head is expected and when the universally quantified variable occurs in the theorem head. A second means of extension comes with the possibility of providing to the tactic the value to instanciate variables with, using a syntax of the form:

`tac thm with` $\mathtt{x}_1$ `:=` val_1 $\mathtt{x}_2$ `:=` val_2`.`

While only certain head-oriented tactics have a way to infer the exact instance of the head that is needed and can apply the first method, all head-oriented tactics can be augmented to provide the second method.

The main example of head-oriented tactics in the Coq system [13] is `Apply`. For a goal C and a theorem `thm` whose head is B and whose depth is n, partitionned in m universal quantifications and p implications, this tactic will detect if the m universally quantified variables occur in B, if C is an instance of B for some assignment of values to these variables, and produce p new goals. The initial hole will be replaced by a term `thm` $a_1 \cdots a_n$ where the a_i's are values for the universally quantified variables or holes corresponding to instances of the left-hand sides of implications.

For instance, let us consider the current goal of interest is the following statement:

$$P(a_1, \ldots, a_{k-1}, b_k, a_{k+1}, \ldots, a_n)$$

and let us suppose the theorem `thm` has the following statement (i.e., it has the following type):

$$\forall x \in A. Q_1(x) \Rightarrow \cdots \Rightarrow Q_m(x) \Rightarrow P(a_1, \ldots, a_{k-1}, x, a_{k+1}, \ldots, a_n).$$

In this situation, the tactic `Apply thm` will infer that the theorem can be applied, that it must be applied to the term b_k and to proofs of $Q_1(b_k)$, ..., $Q_m(b_k)$.

Using the `with ...:= ...` construct to specify the value of some variables does not preclude relying on matching to infer the other values. For instance if the theorem has the following statement:

$$\texttt{le\_trans} : \forall x, y, z : \texttt{nat}.\ x \leq y \Rightarrow y \leq z \Rightarrow x \leq z$$

then the tactic `Apply le_trans with y := some_value` will apply successfully whenever `some_value` is defined and the goal has the form $a \leq b$.

3 Tactic simplification rules

In this section we shall review the various "equivalences" that can be used to simplify tactics. We put the word between quotes because strict equivalence will not always be respected, in the sense that we will also consider transformations where a tactic is replaced with a tactic that does not perform exactly the same operations.

One could view these simplifications as optimizations. It is then sensible to describe what are the optimization criteria. Paradoxically, the main criterion is not the efficiency of the produced code tactic code: this efficiency varies little between the original and simplified expressions. The main criterion is conciseness. All the simplifications we will propose reduce the size of the manipulated tactics, where the size is the number of nodes in the tree structure representing th tactics (this tree structure is an abstract syntax tree).

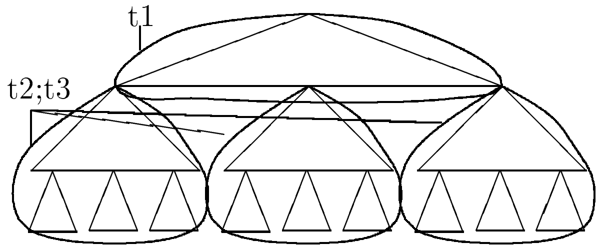
3.1 Flattenning sequences

The `THEN` operator accepts arbitrary tactics as children. For this reason a `THEN` operator can itself appear as child of another `THEN` operator. From an abstract point of view, one may want to consider the algebraic properties of the `THEN` operator, when considered as a binary operator on an abstract space. The first interesting property is that the `THEN` operator is associative.

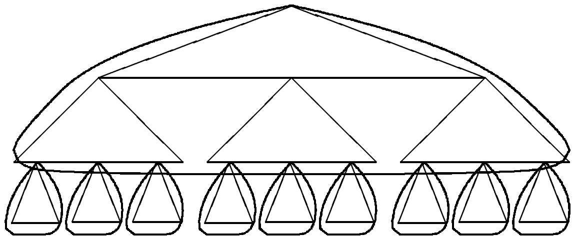
To understand a justification of this property, one needs to recall the intuitive interpretation of tactics as proof term constructors, where the only input is a subgoal, that is a subterm type. The proof term can be abstractly understood as a tree, where each node is decorated with a goal. It is possible to partition the proof tree construction obtained by the successful execution of a tactic of the form:

$$\texttt{tac}_1\texttt{;(tac}_2\texttt{;tac}_3\texttt{)}$$

The goals produced by $\texttt{tac}_1$ are then handled by applications of the tactic $\texttt{tac}_2\texttt{;tac}_3$. This construction can be represented the following way:



It is also possible to partition the structure of the proof term differently, with a divide placed between applications of $\texttt{tac}_2$ and applications of $\texttt{tac}_3$, as represented in the following figure:



Thus, the same proof tree can be viewed as the result of the following tactic:

$$\texttt{(tac}_1\texttt{;tac}_2\texttt{);tac}_3.$$

Based on this remark, the THEN operator is an operator with a list arity: it can accept an arbitrary number of children. With this approach, the associativity property gives the following simplification rule:

$$\texttt{tac}_1\texttt{;}\ldots\texttt{;tac}_n\texttt{;(tac}'_1\texttt{;}\ldots\texttt{;tac}'_p\texttt{)} \rightarrow \texttt{tac}_1\texttt{;}\ldots\texttt{;tac}_n\texttt{;tac}'_1\texttt{;}\ldots\texttt{;tac}'_p \quad (1)$$

3.2 Interactions between THEN and THENL in the Coq system

The use of the same separator ";" for the THEN and THENL tactics gives a special meaning to the question of how the THEN constructor associates: in the presence of a tactic construction of the form:

$$\texttt{tac; tac}'\texttt{;[tac}_1\texttt{|}\ldots\texttt{|tac}_n\texttt{]},$$

one may wonder how the set of tactics $\texttt{tac}_1$, ..., $\texttt{tac}_n$ should be adapted to tac and $\texttt{tac}'$. The solution that is used as a reference in this paper, which corresponds to the implementation in Coq, is that n has to be the number of goals produced by the tactic $\texttt{tac; tac}'$ as a whole. This is imposed by the parser, which naturally binds on the left.

On the implementation side, the structure parsed when recognizing the given text has a tree structure that does not use a THENL node, but only a special kind of element in THEN sequences, to represent the $\texttt{[tac}_1\texttt{|}\ldots\texttt{|tac}_n\texttt{]}$, which we shall call a PARALLEL node. As an unfortunate side effect, the associativity rule is broken. The simplification rule number (1) given in the previous section can only apply if $\texttt{tac}'_p$ is not a PARALLEL node.

3.3 Removing implication elimination

The tactic `LApply` is designed to apply functional theorems to one argument at a time, while head-oriented tactics are designed to apply theorems to many arguments, in fact, as many as possible. When constructing tactics, one usually prefers using direct head-oriented tactics and avoids using the `LApply` tactic, since the first solution yields more concise data.

Tools that use a step-by-step approach to recursively construct tactics (like proof-by-pointing [2]) have a tendency to privilege sequences of uses of the `LApply` tactic. In this section we will study the patterns using compositions of `LApply` and head oriented tactics that can be replaced with simple uses of head oriented tactics.

Let us consider the action of the following tactic, where `tac` is a head-oriented tactic:

$$\texttt{LApply H1;[Intro H;tac H;[tac}_1\texttt{|...|tac}_n\texttt{] | tac}'\texttt{]}$$

Let us suppose that the initial goal is some formula C. By the definition of `LApply`, `H1` must have the form $A \Rightarrow B$ and `H` must have the form `B`. Also, the goal attacked by the tactic `tac H1` in the first case must still be C. Now, `tac H` will produce n goals $C_1, \ldots, C_n$,which are themselves attacked by the tactics $\texttt{tac}_1$ to $\texttt{tac}_n$. The `LApply H1` tactic also produces a goal `A`, which is attacked by the tactic $\texttt{tac}'$.

Now if we consider the action of the following tactic on the same goal:

$$\texttt{tac H1;[tac}'\texttt{|tac}_1\texttt{|...|tac}_n\texttt{]}$$

By the definition of head-oriented tactics, the tactic `tac H1` should succeed and produce the $n+1$ goals $A, C_1, \ldots, C_n$, which are attacked by $\texttt{tac}'$, $\texttt{tac}_1, \ldots, \texttt{tac}_n$. Everything should work smoothly, except that in the first case the tactics $\texttt{tac}_1, \ldots, \texttt{tac}_n$ work in a context where there exists a local `H1` with type B, while no such object is present in the context in the second case. Aside from these differences, this leads us to a simplification rule with the following form:

$$\begin{array}{l} \texttt{LApply}\ H\texttt{;[Intro}\ H_1\texttt{;} \\ \quad \texttt{tac}\ H_1\texttt{;[}tac_1\texttt{|}\ldots\texttt{|}tac_n\texttt{]} \\ \qquad\qquad\qquad\qquad \texttt{|}tac'\texttt{]} \\ \qquad\qquad\qquad\qquad\qquad \to \texttt{tac}\ H\texttt{;[}tac'\texttt{|}tac_1\texttt{|}\ldots\texttt{|}tac_n\texttt{]} \end{array} \tag{2}$$

The two sides of the rule are not strictly equivalent in the following two ways:

1. The goals produced by the two tactics are not produced in the same order in both cases.
2. The context available in new subgoals is not the same in both case.

These discrepancies limit the conditions in which the simplification rule can be applied. Because of the first one, the simplification cannot be applied on a tactic if this tactic is in a context where the order of produced goals is important. For instance, this simplification should not be used to clean a script composed of a sequence of tactics, where each tactic is applied to a goal given by its rank (the usual form of proof scripts in systems like Isabelle, HOL, PVS, Coq, etc.).

Because of the second one, the simplification should not be used without a global check that the cancelled proof term will not be used in sub-tactics. It is not only necessary to check that the cancelled name does not occur again, but also that no tactic that does a systematic search in the local context is used.

In the case of tactics generated using an interactive tool, these constraints on the applicability of the simplification rule can be relaxed for two reasons. First, the generated tactic is usually the last one of the script at that moment so that there is no problem of coherence with respect with the order of new subgoals. Second, the interactive tool may be designed so that names introduced using `LApply` and `Intro` are used only once, thus ensuring that the cancelled name is not used somewhere else (for instance, proof-by-pointing is designed in this manner).

3.4 Compacting *forall* elimination

Forall elimination denotes the process to go from a context containing a formula $\forall x.P(x)$ to a context containing a formula $P(a)$ for some a. If `H` is a proof term having the first formula as type, forall elimination can be performed using the following tactic:

$$\texttt{Generalize (H } a\texttt{);Intro H1}$$

Now, if $P(a)$ is yet another universally quantified formula, one could perform a second forall elimination by putting a second instance of the same pattern behind the `Intro H1` tactic, with a new value b. However, one should notice that `(H a)` is itself a proof term having $P(a)$ as type and the two successive forall eliminations could also be performed correctly by the following tactic:

$$\texttt{Generalize (H } a\ b\texttt{);Intro H2}$$

This explanation can be generalized to an arbitrary number of successive forall eliminations, thus yielding the following simplification rule:

$$\begin{array}{r} \texttt{Generalize (}H\ a_1\ \ldots a_n\texttt{);} \\ \texttt{Intro } H_1\texttt{;} \\ \texttt{Generalize (}H_1\ a_{n+1}\texttt{);} \\ \texttt{Intro } H_2 \rightarrow \texttt{Generalize (}H\ a_1 \ldots a_n\ a_{n+1}\texttt{);} \\ \texttt{Intro } H_2 \end{array} \tag{3}$$

Of course, the general applicability is not granted for this rule, as the same constraints as above appear for names that have been cancelled.

3.5 Mixing $\forall$ and $\Rightarrow$ elimination with head-oriented tactics

Since head-oriented tactics also take a proof term as argument, rule number (3) is directly adaptable to each head-oriented tactic `tac`:

$$\begin{array}{r} \texttt{Generalize (}H\ a_1\ \ldots a_n\texttt{);} \\ \texttt{Intro } H_1\texttt{;} \\ \texttt{tac } H_1 \rightarrow \texttt{tac (}H\ a_1 \ldots a_n\texttt{)} \end{array} \tag{4}$$

Obviously, the tactic `tac` H_1 on the left hand side may itself result from the compaction of implication eliminations, as performed using rule number (2). So rules (2) and (4) can be used to compact uses of head-oriented tactics, implication eliminations, and forall eliminations in one command. But this only works if the forall quantifications are outside the implications.

A pattern where an implication is outside a forall quantification like $A \Rightarrow \forall x.B$ yields a tactic pattern with the following form:

```
LApply H;[Intro H1; tac (H1 a1)|Idtac]
```

It is not possible to replace this tactic with an instance of `tac` applied directly to some proof term. This proof term would have to contain a complete proof of A. However, there is one feature we have described in head-oriented tactics that we have not yet exploited: the possibility to give explicitly variables values, value by value. In the case above, the relevant tactic is:

```
tac H1 with x := a1
```

This remark does not carry over very well into a simplification rule. Not all the data present in the simplification result appears in the reduced expression: the variable name `x` is produced only when one knows that the forall quantification in the statement (the type of `H`) introduces a variable of that name. We are confronted with the fact that the tactic pattern which we use to perform forall elimination does not contain this name and the information is lost.

To cope with this issue, we assume that the algorithm that produces the tactic expression is capable of annotating the generated tactics with variable names. From now on, we will consider forall-elimination patterns of the following form, indicating that the forall quantification introduced a variable named `x`:

$$\texttt{Generalize (H a}^{x}\texttt{);Intro H1}$$

With this approach, rule number (3) should be replaced with the following rule:

$$\begin{array}{l}\texttt{Generalize (}H\ a_1^{x_1}\ \ldots a_n^{x_n}\texttt{);}\\ \qquad\qquad\texttt{Intro } H_1\texttt{;}\\ \quad\texttt{Generalize (}H_1\ a_{n+1}^{x_{n+1}}\texttt{);}\\ \qquad\qquad\texttt{Intro } H_2 \rightarrow \texttt{Generalize (}H\ a_1^{x_1}\ldots a_n^{x_n}\ a_{n+1}^{x_{n+1}}\texttt{);}\\ \qquad\qquad\qquad\qquad\texttt{Intro } H_2\end{array} \tag{5}$$

Rule number (4) should also be modified accordingly.

Now the following rule makes it possible to switch from a simple use of head-oriented tactics to a more elaborate one:

$$\texttt{tac (}H\ a_1^{x_1}\ a_n^{x_n}\texttt{)} \rightarrow \texttt{tac } H \texttt{ with } x_1 \texttt{ := } a_1 \ldots x_n \texttt{:=} a_n \tag{6}$$

The right hand side of this rule is arguably less concise than its left hand side, so it should be used with care. In fact, it should be used only when it makes it possible to use the following rule, corresponding to an implication elimination preceding an arbitrary mixture of forall and implication eliminations and a head-oriented tactic.

$$\texttt{LApply } H\texttt{;[Intro } H_1\texttt{;}$$

$$
\begin{array}{l}
\texttt{tac}\ H_1\ \texttt{with}\ x_1 := a_1 \\
\qquad\qquad\quad \ldots \\
\qquad\qquad\quad x_n := a_n \\
\qquad\qquad\quad \texttt{|Idtac]} \rightarrow \texttt{tac}\ H\ \texttt{with}\ x_1 := a_1 \ldots x_n := a_n
\end{array}
\tag{7}
$$

There is a similar rule for a forall elimination preceding an arbitrary mixture of forall and implication eliminations and a head oriented tactic.

$$
\begin{array}{l}
\texttt{Generalize (}\ H\ a^x\texttt{);} \\
\qquad \texttt{[Intro}\ H_1\texttt{;} \\
\quad \texttt{tac}\ H_1\ \texttt{with}\ x_1 := a_1 \\
\qquad\qquad\quad \ldots \\
\qquad\qquad\quad x_n := a_n \\
\qquad\qquad\quad \texttt{|Idtac]} \rightarrow \texttt{tac}\ H\ \texttt{with}\ x := a\ x_1 := a_1 \ldots x_n := a_n
\end{array}
\tag{8}
$$

4 Conclusion

The work described in this paper presents a few classes of tactics and tactic constructors (tacticals) and simplification rules between tactic patterns. These simplifications are used as the formal basis for an simplifier that has actually been implemented in a proof environment composed of a user-interface with graphical features [1], where tactics can be edited by hand or produced mechanically from mouse-clicks issued by the user. Adding this simplifier to the command generator is very well perceived by the user, even though the optimized commands are not strictly equivalent to the original ones. One positive feature is that generated commands look like hand-crafted ones: this makes them more readable.

The work presented here has some limitations. First, we have deliberately avoided a setting where logical formulas can contain holes (existential variables). In such a context, many of the simplifications we have presented become very fragile. For instance, our simplifications rely on the fact that direct subgoals can be attacked in any order. This is no longer true in the presence of existential variables.

In this paper, we have chosen to only introduce informally the semantics of each tactical and the characteristics of the class of head-oriented tactics. To completely justify the validity of our simplification, one would need a much more formal description of tactics. This could be a complementary area of research, which we have started with an abstract description of tactics and tacticals, developed directly in the proof system. Thus, we may even be able to prove mechanically the validity of our simplifications.

Another area of possible work is to understand whether the `THEN` and `THENL` constructs provide the better set of basic constructs for writing complex tactics. In some respect, it seems that a variant, mixing the semantics the two constructs could make it easier to write shorter tactics. For example a composition tactical could differentiate the new subgoals in two sets: those corresponding to the tactic applied to the head (principal subgoals, in some —etymological— sense) and those corresponding to implication antecedents due to head-oriented behavior (side goals).

Acknowledgements

The work on proof-by-pointing, which initiated this one, was started with Laurent Théry and Gilles Kahn, it also was their adamant pressure for better usage that led to these simplifications. The work described in this paper is the explanation of algorithms implemented in the CtCoq system. I would like to thank the other developers of this system, Janet Bertot and Francis Montagnac for their help in this development.

References

1. Janet Bertot and Yves Bertot. CtCoq: A system presentation. In *Automated Deduction (CADE-13)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 231–234. Springer-Verlag, July 1996.
2. Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160, 1994.
3. Yves Bertot, Thomas Schreiber, and Dilip Sequeira. Proof by pointing in a weakly structured context. ftp://ftp.dcs.ed.ac.uk/pub/lego/pbp/main.ps.
4. Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. To appear in Journal of Symbolic Computation.
5. Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. Centaur: the system. In *Third Symposium on Software Development Environments*, 1988. (Also appears as INRIA Report no. 777).
6. Debra Cameron and Bill Rosenblatt. *Learning GNU Emacs*. O'Reilly & Associates, Inc., 1991.
7. Robert Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harber, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.
8. Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, University of Pennsylvania, 1989.
9. Amy Felty and Douglas Howe. Tactic theorem proving with refinement-tree proofs and metavariables. In *Automated Deduction(CADE-12)*, volume 814 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1995.
10. Amy Felty and Laurent Théry. Interactive theorem proving with temporal logic. To appear in Journal of Symbolic Computation.
11. Michael Gordon and Thomas Melham. *Introduction to HOL : A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
12. Timothy Griffin. *Notational definition and top-down refinement for interactive proof development systems*. PhD thesis, Cornell University, 1988.

13. INRIA. *The Coq Proof Assistant Reference Manual*, December 1996. Version6.1.
14. The LEGO World Wide Web page. url `http://www.dcs.ed.ac.uk/home/lego`.
15. Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 213–237. Springer-Verlag, 1994.
16. Per Martin-Löf. *Intuitionistic type theories.* Bibliopolis, 1984.
17. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990. ISBN 0-262-63132-6.
18. Lawrence C. Paulson. *Logic and computation, Interactive proof with Cambridge LCF.* Cambridge University Press, 1987.
19. Lawrence C. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.
20. Lawrence C. Paulson and Tobias Nipkow. *Isabelle : a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.
21. Laurent Théry. A proof development system for the HOL theorem prover. In *Higher Order Logic theorem proving and its applications*, volume 780 of *Lecture Notes in Computer Science.* Springer-Verlag, August 1993.
22. Laurent Théry, Yves Bertot, and Gilles Kahn. Real Theorem Provers Deserve Real User-Interfaces. *Software Engineering Notes*, 17(5), 1992. Proceedings of the 5th Symposium on Software Development Environments.