A Tactic Language for Declarative Proofs *

Serge Autexier and Dominik Dietrich

German Research Center for Artificial Intelligence (DFKI), Bremen, Germany
{autexier|dietrich}@dfki.de

Abstract Influenced by the success of the MIZAR system many *declarative proof languages* have been developed in the theorem prover community, as declarative proofs are more readable, easier to modify and to maintain than their *procedural* counterparts. However, despite their advantages, many users still prefer the procedural style of proof, because procedural proofs are faster to write. In this paper we show how to define a *declarative tactic language* on top of a *declarative proof language*. The language comes along with a rich facility to declaratively specify conditions on proof states in the form of *sequent patterns*, as well as *ellipses* (dot notation) to provide a limited form of iteration. As declarative tactics are specified using the declarative proof language, they offer the same advantages as declarative proof languages. At the same time, they also produce declarative justifications in the form of a declarative proof script and can thus be seen as an attempt to reduce the gap between procedural and declarative proofs.

1 Introduction

The development of interactive tactic based theorem provers started with the LCF system [19], a system to support automated reasoning in Dana Scotts "Logic for Computable Functions". The main idea was to base the prover on a small trusted kernel, while also allowing for ordinary user extensions without compromising soundness. For that purpose Milner designed the functional programming language ML and embedded LCF into ML. ML allowed to represent subgoaling strategies by functions, called *tactics*, and to combine them by higher order functions, called *tacticals*. By declaring an abstract type *theorem* with only simple inference rules type checking guaranteed that tactics decompose to primitive inference rules.

While allowing for efficient execution of recorded proofs by representing them as a sequence of tactic applications, it has been recognized that these kind of proofs are difficult to understand for a human. This is because the intermediate states of a proof become only visible when considering the changes caused by the stepwise execution of the tactics. Tactic proofs can be extremely fragile, or reliant on a lot of hidden, assumed details, and are therefore difficult to maintain and modify (see for example [34] or [20] for a general discussion). As the only information during the processing of a proof is the current proof state and the next tactic to be executed, a procedural prover has to stop checking at the first error it encounters.

* This work was supported by German Ministry for Research and Education (BMBF) under grant 01 IW 07002 (project FormalSafe).

This has led to *declarative proof languages*, inspired by MIZAR [29], where proof steps state *what* is proved at each step, as opposed to a list of interactions required to derive it. It has been argued that structured proofs in a declarative proof language are easier to read and to maintain. Moreover, as a declarative proof contains explicit statements for all reasoning steps it can recover from errors and continue checking proofs after the first error. It has been noted in [32] that a proof language can be implemented rather independently of the underlying logic and thus provides an additional abstraction layer. Due to its advantage many interactive theorem provers nowadays support declarative proofs (see for example [28,32,3,11]).

Another motivation for a declarative proof language comes from a research community dealing with the integration of proof assistants into development environments and supporting the so-called *document centric approach* [16,2]. The main idea is that the document, containing a formal theory and the formal proofs, is the central medium around which tools to assist the author are made available. As many tactics are developed while developing the formal theory, it is only consequent to integrate them into the document. Currently, this is not the case, as tactics are usually written in the underlying programming language of the prover.

Contributions. In this paper we present a *declarative tactic language* on top of a *declarative proof language* (which can be seen as an extension of [14]). To our best knowledge, such a language has not been presented before. Our language comes along with a rich facility to declaratively specify proof states (and conditions on them) in the form of *sequent patterns*, as well as *ellipses* (dot notation) to provide a limited form of iteration. The language is intended to provide a simple to use tactic language layer to bridge the gap between the predefined proof operators and the programming language of the proof assistant. This new layer of abstraction can be seen in analogy to what has been done by introducing declarative proof languages. We believe that declarative tactic languages offer similar advantages as declarative proof languages, namely robustness and readability, and that the trend towards declarative proof languages will carry on with declarative tactic languages. Interestingly, the trace of a declarative tactic is a declarative proof script and can thus be inserted into the document if desired. Moreover, because of its additional abstraction, it might provide possibilities to exchange reasoning procedures between different proof assistants in the long-term view.

The structure of the paper is as follows: Section 2 gives a more detailed background and motivates our language by means of a simple example. Section 3 presents the basic proof script language. Section 4 motivates and extends the language by an ellipsis construct. Finally, we conclude the paper in Section 5 with a discussion of related work.

2 Background and Introductory Example

At present, two main formalization styles are supported by interactive theorem provers, namely the *procedural style* and the *declarative style*.

In the procedural style, proofs consist of a sequence of tactic applications, as shown on the left of Figure 1. Intuitively, the proof script corresponds to the edges of a derivation tree being labeled with the tactic names. Even though there are different implementations of the notion of a tactic, each of these tactics can be understood as a program

```
theorem natcomp: "(a::nat) +
(b::nat) = (b::nat)+(a::nat)"
apply (induct a)
apply (subst add_0)
apply (subst add_0_right)
apply (rule refl)
apply (subst add_Suc_right)
apply (subst add_Suc)
apply (simp)
done
```

```
theorem natcomplus: "(a::nat) + b = b+a"
proof (induct a)
  show "0 + b = b + 0"
  proof (-)
    have "0+b=b" by (simp only: add_0)
    also have "...=b+0" by (simp only:
add_0_right)
    finally show ?thesis .
  qed
  next ...
```

Figure 1: Procedural and declarative proof script in ISABELLE/ISAR

taking a list of goals together with a justification function as input and returning a new set of goals together with an updated justification function. The justification function is an internal function in the underlying programming language, such as ML, and cannot be presented to the user. By being a sequence of explicit program calls, a procedural proof contains explicit statements of *what to do*. In particular, the reader does not see the proof state unless he executes the tactic. Therefore, (procedural) proofs are considered not to be human readable and difficult to maintain. Procedural tactics are usually written in the underlying programming language of the assistance system, such as ML, conflicting with the document centric approach.

In the declarative style, proofs consist of structured blocks, where each block consist of a list of statements, connected by a fixed set of keywords. The statements specify what is proved at each step. Intuitively, a declarative proof script thus corresponds to the information contained in the nodes of a derivation tree. Most declarative languages require the user to give hints justifying the statement using previous statements. However, in principle a declarative proof can simply be a sequence of intermediate assertions, acting as islands or step stones between the assumptions and the conclusion (by omitting the constraints indicating how to find a justification of the proof step) leaving the task of closing the gaps to automation tools. Such islands are sometimes also called *proof plans* [13] or *proof sketches* [33]. Surprisingly, quite many systems – sometimes called *proof finders* or *proof planner* – have been developed trying to automatically close such gaps, such as MIZAR, NQTHM [8], the SPL system [34], the SAD system [30], the NAPROCHE [24] system, the SCUNAK system [9], TUTCH [1], or ΩMEGA [15].

The main advantage of a declarative proof script is that intermediate assertions are shown in the proof script. While this makes the proof easier to read, it makes it more difficult to write, as the proofs tend to be longer (see the example on the right of Figure 1). As a consequence, in practice users often prefer the procedural style of proof.

2.1 From Declarative Proof Scripts to Declarative Tactics

Having the correspondence between procedural and declarative proofs in mind and recalling that in the simplest case a tactic is a sequence of inference applications, representing a partial proof, it appears suggestive to think about declarative tactics as analog

```
theorem natcomplus: (a::nat) +
(b::nat) = b+a
proof
  subgoals by (induct a)
    subgoal 0 + b = b + 0
    subgoal Suc a + b = b + Suc a
      using IH:a+b=b+a
  end
qed
```

```
theorem natcomplus: (a::nat) +
(b::nat) = b+a
proof
  subgoals by (induct b)
    subgoal a + 0 = 0 + a
    subgoal a + Suc b = Suc b + a
      using IH: a+b=b+a
  end
qed
```

Figure 2: Declarative proof with gap resulting by induction over a, respectively b

to procedural tactics. In contrast to procedural proofs the justification should be declarative, but we additionally require it to be specified using a declarative language, namely the proof language itself.

Consider for example the problem in Peano arithmetic of showing the commutativity of addition, that is, $a+b=b+a$. Of course, a proof can easily be generated in the procedural style. However, because of its advantages what we are really interested in is a declarative proof. Starting by induction over one variable, two possible proof attempts in ΩMEGA-proof language [15] are shown in Figure 2. Note that as the proof is still partial, it contains unjustified statements and can thus be seen as a proof sketch or a proof plan. This is similar to the "gap" command introduced in [16].

To automate the generation of one of these scripts, three steps are necessary. First, we need the *control information* over which variable the induction has to be performed. This is for example possible by analyzing the universally quantified variables and preferring those in recursion position. Second, we need a *schematic proof script* (a proof script with schematic variables), as well as a mechanism to instantiate schematic variables with actual terms, which is in our case the desired induction variable. Indeed, by comparing the scripts above, we observe that the two proof scripts can be made equal by replacing the induction variable by a schematic variable. The choice point over the induction variable can be expressed as membership in a (sorted) list of admissible induction variables, which can be easily computed in the underlying programming language of the prover. Finally, to be able to perform induction over the natural numbers on different problems, we replace the instance of the problem by a schematic variable, and use matching against the proof state to establish their relation.

```
strategy natinduct
 cases * ⊢ φ
    with x in (analyzeinductvars "φ")
      P=(abstract "φ" "x") ->
proof
  subgoals by (induct x)
    subgoal P 0
    subgoal P (suc x) using IH: P x
  end
qed
```

Figure 3: Declarative induction tactic

Our realization is shown in Figure 3, where we use quotes to refer to expressions in the tactic language within the underlying programming language. To illustrate the different levels of the tactic, we shade the background of expressions in the tactic language, while expressions in the proof language are unshaded. Expressions in the underlying programming language are written in sans serif font.

Even though the result of executing a declarative proof script on a proof state is again a proof state – as in the case of procedural tactics –, it provides a simple mechanism to view the proofs at different levels of granularity, namely either as a single tactic invocation in the style of ... **by** `name`, but also by showing the declarative proof script obtained by replacing the schematic variables by the terms computed by the tactic. Note that one reason why some users favor the procedural style of proofs over the declarative style of proofs is that the procedural style is faster to type. This benefit remains when invoking declarative tactics. However, we additionally obtain a declarative proof script. In that sense, they can be seen as a means to close the gap between the procedural style and declarative style of proofs.

3 Development of the Language

In the simplest form, a declarative tactic is a (partial) proof in the proof language. For more complex situations, we have to answer the following questions: (i) When is the tactic applicable? (ii) How do the intermediate proof states (islands) look and how can they be generated? (iii) What is the justification for the statements?

To declaratively specify (i), we provide the **cases** construct and matching facilities to relate schematic variables with a given proof state and to restrict the applicability of the tactic (see Figure 4 for the grammar rules and Figure 3 for an example). The matcher specifies a matching condition on the proof state in the form of a sequent. * is used to indicate that the length of the antecedent of the sequent can be higher than the length of the antecedent of the matcher. **[***t***]** denotes the condition that t occurs as a subterm in a formula of the sequent. Subterm occurrences can further be restricted by the specification of polarities, where we use + to indicate a positive subformula and - to indicate a negative subformula. By using a schematic variable instead of +, -, the polarity is accessible within the tactic via that variable. Note that in general a matcher can match a given sequent in different ways and thus introduces nondeterminism.

defstrat	::= **strategy** *name stratexp*
stratexp	::= **cases** (*matcher stratexp*) $^+$
	| *proof* (**with** *assignments*)?
matcher	::= *matchhead whereexp*?
	(**with** *assignments*)
whereexp	::= **where** *prog*
matchhead	::= *sequent* | var
sequent	::= *termpatterns* (**,***)? $\vdash$ *termpattern*
termpatterns	::= *termpattern*
	| *termpatterns* **,** *termpattern*
termpattern	::= *form* | **[** *term* **]** (^ *termqualifier*)?
termqualifier	::= $+$ | $-$ | *var*
assignments	::= *lhs assignop prog*
assignop	::= **=** | **in**
lhs	::= *form* | **(** *form* (, *form*)$^+$ **)**

Figure 4: Basic Tactic Language

(ii) is expressed within the proof language, while we allow the statements to contain schematic variables. Figure 5 shows the abstract syntax of our proof language, which is standard except that metavariables[1] are allowed in the statements. Metavariables can be

[1] Metavariables are supported by the underlying ΩMEGA prover and are not to be confused with the schematic variables

proof	::= **proof** *steps* **qed**	*assume*	::= **assume** *steps from* **thus** *form*
steps	::= (*ostep*;*steps*)|*cstep*	*fact*	::= *sform* | *by from*
ostep	::= *set*|*assume*|*fact*|*goal*	*goals*	::= **subgoals** (*goal*)$^+$ *by*
cstep	::= *trivial* | *goals*|*cases*|ε	*cases*	::= **cases** (*form* { *proof* })$^+$ *by from*
by	::= **by** *name*? | *proof*	*goal*	::= **subgoal** *form* (**using** *form* (**and** *form*)$^+$)? *by*
from	::= **from** (*label* (, *label*)*)?	*set*	::= **set** *var*=*form* (**,** *var*=*form*)*
sform	::= *form* | **.** *binop form*	*trivial*	::= **trivial** *by from*

Figure 5: ΩMEGA proof script language

instantiated using the **set** construct. The **subgoals** construct performs an explicit backward step. Each new subgoal is stated by a **subgoal**, followed by a proof of that subgoal. New assumptions for that subgoal are introduced within the **using** form. If only a single subgoal is introduced, the keyword **subgoals** can be omitted and the subsequent proofs refers to the justification of the reduction.

The value of schematic variables is computed during the expansion of tactic. The grammar rules for tactics are depicted in Figure 4. Here, *form* and *var* are from the underlying term language with possible labels on subterms, such as $(L1 : A) \wedge (L2 : B)$ and schematic variables. We use $\perp$ to indicate failure. To allow for a limited form of non-determinism we provide the assignment operator **in**, which chooses from a list of possibilities. As we cannot expect to provide a fixed language to express metalevel conditions and to perform metalevel analysis (in our case the extraction and sorting of the admissible induction variables), a reasonable strategy is to link-in the underlying programming language of the prover here, indicated in the grammar by *prog*.

For (iii), the justification is either underspecified (no **by** and no **from**), partially specified (**by** and/or **from**), or fully specified (subproof given).

These extensions already allow us the specification of the induction tactic in a declarative form (see Figure 3). Note that schematic variables can be used as placeholders for arbitrary terms, in particular terms generated by an oracle without justification. This provides a convenient means to integrate results from external systems, such as computer algebra systems (CAS) (see [10] for an overview for combining CAS systems and theorem provers). In such a case, we can either leave the justification of the oracle step underspecified (gap), or indicate a tactic to be used to justify it.

An example of such a tactic is given in Figure 6. The tactic is applicable if the goal has the form `abs(GOALLHS)<GOALRHS` and calls a CAS to factor `GOALLHS`. To that end, the schematic variable `Y` is bound to the result of the factorization provided by MAXIMA, where the translation of the term `GOALLHS` into the syntax of MAXIMA and the translation back is internalized in `maxima-factor`. If this succeeds, the script specified in the **proof** ... **qed** block is instantiated and inserted. Being executed, it reduces the goal `abs(GOALLHS)<GOALRHS` to the goal `abs(Y)<GOALRHS`. This reduction is justified by (1) showing the equality between `Y` (the factorization provided by the CAS) and `GOALLHS`, and then applying the fact that `abs` is a function. Note that the same tactic is also expressible in a forward style by relying on **assume** and that all labels in the proof script are generated at runtime and are renamed if already present in the context.

```
strategy maximafactorabs
cases
  * |- ((abs(GOALLHS)) < GOALRHS) ->
  proof
    subgoal abs(Y) < GOALRHS by
    proof
      L2:(Y = GOALLHS) by abeliandecide
      L3: abs(Y) = abs(GOALLHS) by (f=abs in arg_cong) from L2
      trivial from L3
    qed
  qed
  with Y = (maxima-factor "GOALLHS")
```

Figure 6: Call of a CAS to factor a subterm of the goal formula

Semantics. Figure 7 shows the semantics of our language constructs by showing how to expand a declarative tactic to a declarative proof script. The expansion mechanism works on configurations $\langle PS;\Gamma;exp\rangle$, where PS denotes the current proof state, and Γ a context, which is initially empty and keeps track of bindings for schematic variables. exp denotes the expression to be expanded. Configurations evaluate either to a proof script, denoted by the relation $\hookrightarrow$, or to an environment, denoted by the relation $\rightarrow$. We use the notation $\Gamma \cup a = b$ to denote the update of Γ with the binding $a = b$, and the symbol $\perp$ to denote failure. To keep the rules simple, some rules are non-deterministic. In the actual implementation, of course, all results are lazily produced and stored for backtracking. $\mathsf{instance}(\Gamma, S)$ denotes the instantiation of the schematic proof S by replacing the schematic variables with their values in Γ. It is only applicable if all schematic variables are bound. We use eval to evaluate a LISP expression *prog*; the sequent matching is abstracted in the function match (which is also non-deterministic).

To enhance readability, we have grouped corresponding rules together. The first group describes the expansion of the **cases** construct, which returns the result of the first case that succeeds. An individual case is either a proof script (second group), or of the form *matchhead (***where** *exp)*$^?$ (third group). The value of schematic variables is computed within the **with** construct (see the last group) which uses eval to evaluate expressions of the underlying programming language. Sequent matching works by first invoking the matcher on the current proof state and then evaluating additional condition.

4 Extension of the Basic Language

So far, our declarative tactic language is less expressive than its procedural counterpart. As a matter of fact, there exist powerful procedural tactics using for example the constructs of loops, such as simplification, which are per se difficult to express declaratively, as we cannot expect to determine their result unless we execute it. While this is unproblematic when using them to close gaps between intermediate statements, their treatment as black boxes makes it difficult to express how to process their results further in the form of a continuation, because the structure of the formula is lost. For example, all we know about the result term of factorization in Figure 6 is that it is of the form Y.

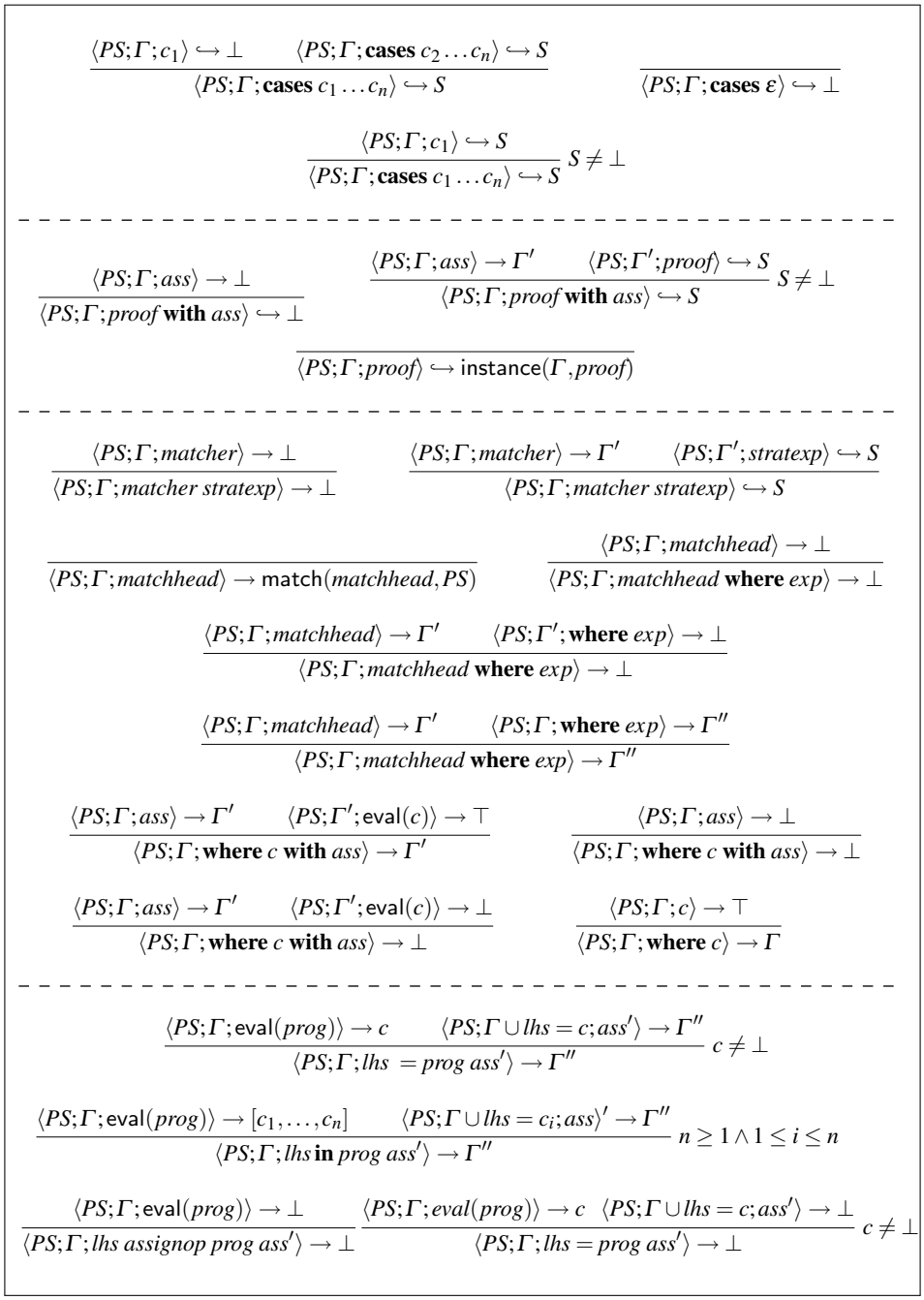

Figure 7: Expansion Rules for a Declarative Tactic

To illustrate a possible continuation of `maximafactorabs`, let us consider the so-called *limit domain* which contains statements about the limit and continuity of functions. It was proposed by Bledsoe [6] as challenge problems for automated theorem provers. The proofs typically involve ε-δ arguments and are interesting because both logic and computation have to be combined to find a solution to the problem given at hand, while still being simple enough to allow for an automatic solution based on heuristics. Several people from the AI community have tackled this domain (see for example [26,4]) One heuristic of the limit domain to bound factors is to reduce the problem that the product $\beta\gamma$ is arbitrarily small to the problem that of showing that β is arbitrarily small and γ can be bounded. This heuristic is called *factor bounding* and described (in [4], p. 77f) as follows :

> "The following rule is stated for simplicity using only two factors, but the rule is implemented for a product of any number of factors.
>
> $$\frac{\Gamma, |\alpha| < \delta \qquad \Gamma, |\alpha| < \delta | \vdash |\beta| < \varepsilon/(M+1)}{\Gamma, |\alpha| < \delta \vdash |\beta\gamma| < \varepsilon}$$
>
> When this rule is implemented, we take M to be a fresh metavariable, and forbid to M all the variables that are forbidden to δ. In the present implementation, the rule is used only when δ is a metavariable."

While our language can already deal with the factorization, we cannot yet declaratively express the factor bounding *for an arbitrary number of factors* within a single tactic and use it as continuation. Moreover, what we really want is to express the factor bounding as the continuation after successful factorization.

Reconsidering the problem we observe that the difficulty is due to the missing information we have about the factorization, namely the number of factors which is dynamic. Interestingly, even though the structure is dynamic, syntactic patterns are commonly used in mathematical practice to capture such a structure, by making use of ellipses (dot notation). In our example, a dynamic number of factors can be expressed by the pattern $X_1 * \ldots * X_n$. Internally, patterns are implemented by subsequently invoking the matcher for pattern X until it fails, taking the associativity of the binary operator into account, resulting in a list of matches which are stored in an internal variable X.

Coming back to our example, we notice that all factors but one factor shall be bounded. Therefore, we need also constructs to dynamically construct statements in the proof script language. To that end, we introduce a **foreach** construct. The grammar for the extended language constructs is shown in Figure 8. Binary patterns (*binop*) can be used in places where previously only *form* was allowed. Step is extended by *foreachstep* construct. Moreover, *assrhs* is extended by the foreach assign-

binoppat	::= *pattern binop* **..** *binop pattern*
listaccess	::= (*listterm* | *var*) _ (*var* | *number*)
listterm	::= *listdel* **..** *listdel*
listdel	::= *var* | *number* | *pattern*
pattern	::= *binoppat* | *listaccess* | *from*
foreachexp	::= **foreach** *var* **in** *listterm* (**where** cond)?
foreachstep	::= *foreachexp* steps **end**
foreachass	::= *foreachexp var* _ *var=prog*

Figure 8: Dynamic matching constructs

```
strategy factorbound
 cases
   abs(LHS)<RHS,* |- abs(GOALLHS) < GOALRHS
   where (and (variable-eigenvar.is "GOALRHS")
              (metavar-is "RHS")
              (some #'(lambda (x) (term= "LHS" "x")) "Y_1 .. Y_N"))
   with Y_1 * .. * Y_N = (maxima-factor "GOALLHS")
        j = (termposition "LHS" "Y_1 .. Y_N")
   ->
   proof
     L1: GOALLHS= Y_1 * .. * Y_N by abeliandecide
     foreach i in 1..N where (not (= "j" "i"))
       Y_j <= MV_j by linearbound
     end
     L2: abs(GOALLHS)=abs( Y_1 * .. * Y_N ) from L1
     .<= abs(Y_1) * .. * abs(Y_N)
     .< MV_1 * .. * MV_N
     .<= GOALRHS
   qed
   with foreach i in 1..N
          M_i = (if (= "i" "j") "RHS" (make-metavar (term-type "RHS")))
```

Figure 9: Dynamic pattern matching and proof script generation

ment (*foreachass*). These extensions will allow us to specify a variant of the factorbound method in a convenient way (see Figure 9 on page 10).

Ellipses. So far our constructs for matching and constructing terms are static in the sense that their actual form was already determined at compile time. For example, a pattern of the form $lhs = rhs$ checks whether the input formula is an equality and binds its first argument to *lhs* and its right argument to *rhs*. *Dynamic Patterns* on the contrary are patterns that capture dynamic structures, such as all elements of a finite list. We support a simple dynamic pattern, an ellipsis for binary operators, written $A\ op \ldots op\ A'$, which acts like a Kleene star, as well as a list pattern which is similar except that *op* is omitted. Internally, such dynamic patterns are represented as lists, whose length is stored in an additional variable. To individually access the lists, we provide an accessor function _. That is, A_n denotes the n-th element in the list A. If n is a variable, then n is called *access variable*. In the current implementation, patterns are restricted to *simple patterns*, which are patterns that unify under a substitution σ whose domain consists only of access variables. Patterns can be used both in conditions, as left hand side of assignments, as well as in proof script terms. Some examples are shown in Figure 10.

The foreach construct provides a simple form of iteration over a list of values obtained from a dynamic pattern. It can be used to construct statements in the proof script language as well as to construct a list of schematic variables. Its expansion rules are shown in Figure 11, grouped into the expansion rules to expand **foreach** within a proof script,

Expression	Meaning
`A_1 + .. + A_N`	finite sum with N summands
`abs(A_1) * .. * abs(A_N)`	product with N factors of the form `abs(_)`
`(X_1 + Y_1) * .. * (X_N + Y_N)`	product of terms of binary sums
`1 .. 5`	list [1,2,3,4,5]
`abs(A_1) .. abs(A_N)`	list with N terms of the form `abs(_)`

Figure 10: Patterns using ellipses

and the expansion rules to expand **foreach** in assignments. Note that in case of assignments a list containing all produced values is constructed, which has always the length of list over which it is iterated. In the case that the condition evaluates to $\bot$ a term *false* is inserted at the corresponding position.

Illustration of the Tactic As an example, we consider the problem of proving $\lim_{x\to 3}\frac{x^2-5}{x-2} = 4$. After expanding the definition of lim, the proof state consists of the two goals $\varepsilon > 0, |x-3| < ?\delta \vdash |\frac{x^2-5}{x-2} - 4| < \varepsilon$ and $\varepsilon > 0 \vdash ?\delta > 0$. The declarative proof script is shown at the top of Figure 12, where the declarative tactic `factorbound` (see Figure 9) is not yet processed.

Processing the `factorbound`-statement expands it and results in the following steps:

1. The pattern of the cases condition is matched, yielding the following binding: $\{\texttt{LHS} \mapsto x-3, \texttt{RHS} \mapsto ?\delta, \texttt{GOALLHS} \mapsto \frac{x^2-5}{x-2} - 4, \texttt{GOALRHS} \mapsto \varepsilon\}$
2. To be able to evaluate the **where** condition, the first **with** part is evaluated. This results in the following factorization: $Y_1 * \ldots * Y_n = (x-3) * (\frac{1}{x-2})(x-1)$. Internally, a list $Y = [(x-3), (\frac{1}{x-2}), (x-1)]$ is generated, n is bound to 3. In the next assignment, and j is bound to 1 by looking up $x-3$ in the list of factors.
3. The conditions of the where part evaluates to true
4. The **with** part of the **proof** is evaluated, generating a list $M = [?\delta, ?MV1, ?MV2]$ of length 3.
5. The **proof** part is expanded and inserted, resulting in the proof script shown at the bottom in Figure 12.

Declarative Tactics and Parameters. For procedural tactics it is often convenient to pass control information in the form of arguments when calling the tactic. For example, in the introductory example we invoked the tactic `induct` with the argument "x" indicating the induction position. A similar mechanism is desirable in the case of declarative tactics. In our language, arguments are treated as schematic variables. If a schematic variable occurs in the proof script, but is neither used in the **cases** construct nor bound within the **with** environment, it corresponds to a required argument. Schematic variables that are computed within the tactic can be passed as optional arguments. In such a case, the passed argument overwrites the computed argument. We provide the common syntax *var*=*value* **in** *tactic*.

5 Conclusion and Related Work

In this paper we presented the construction a declarative tactic language on top of a declarative proof language. Our language comes along with a rich facility to declara-

$$\frac{\langle PS;\Gamma;listterm\rangle \to [e_1,\ldots,e_n] \qquad \langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_1,\ldots,e_n]\ (\textbf{where } c)^?exp_2\rangle \hookrightarrow S}{\langle PS;\Gamma;\textbf{foreach } var \textbf{ in } listterm\ (\textbf{where } c)^?\ exp_2 \textbf{ end}\rangle \hookrightarrow S}$$

$$\frac{}{\langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [](\textbf{where } c)^?\ exp_2 \textbf{ end}\rangle \hookrightarrow \varepsilon}$$

$$\frac{\langle PS;\Gamma\cup var=e_1;exp_2\rangle \hookrightarrow S1 \qquad \langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_2,\ldots,e_n]\ exp_2\rangle \hookrightarrow S2}{\langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_1,\ldots,e_n]\ exp_2\rangle \hookrightarrow S1\ S2}$$

$$\frac{\begin{array}{c}\langle PS;\Gamma\cup var=e_1;exp_2\rangle \hookrightarrow S1\\ \langle PS;\Gamma\cup var=e_1;c\rangle \to \top\end{array} \qquad \langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_2,\ldots,e_n] \textbf{ where } c\ exp_2\rangle \hookrightarrow S2}{\langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_1,\ldots,e_n] \textbf{ where } c\ exp_2\rangle \hookrightarrow S1\ S2}$$

$$\frac{\langle PS;\Gamma\cup var=e_1;c\rangle \to \bot \qquad \langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_2,\ldots,e_n] \textbf{ where } c\ exp_2\rangle \hookrightarrow S2}{\langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_1,\ldots,e_n] \textbf{ where } c\ exp_2\rangle \hookrightarrow S2}$$

$$\frac{\langle PS;\Gamma;listterm\rangle \to [e_1,\ldots,e_n] \qquad \langle PS;\Gamma;\textbf{iterate } ass \textbf{ in } [e_1,\ldots,e_n]\ (\textbf{where } c)^?prog\rangle \to \Gamma'}{\langle PS;\Gamma;\textbf{foreach } var \textbf{ in } listterm\ (\textbf{where } c)^?\ \underbrace{name\_var=prog}\rangle \to \Gamma'}$$

$$\frac{\begin{array}{c}\langle PS;\Gamma\cup var=e_1;ass\rangle \to \Gamma'\\ \langle PS;\Gamma\cup var=e_1;c\rangle \to \top\end{array} \ \langle PS;\Gamma'\backslash(var=e_1);\textbf{iterate } var \textbf{ in } \overline{[e_2,\ldots,e_n]}^{;ass} \textbf{ where } c\ ass\rangle \to \Gamma''}{\langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_1,\ldots,e_n] \textbf{ where } c\ ass\rangle \to \Gamma''}$$

$$\frac{\langle PS;\Gamma\cup var=e_1;c\rangle \to \bot \qquad \langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_2,\ldots,e_n] \textbf{ where } c\ ass\rangle \to \Gamma'}{\langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_1,\ldots,e_n] \textbf{ where } c\ ass\rangle \to \Gamma'}$$

Figure 11: Expansion of the **foreach** construct

tively specify proof states (and conditions on them) in the form of *sequent patterns*, as well as *ellipses* (dot notation) to provide a limited form of iteration. We believe that declarative tactic languages offer similar advantages than declarative proof languages, namely robustness, readability, and maintainability, because intermediate results of the tactic are visible due to the use of the declarative proof language for their specification. In addition to that, the main feature of declarative tactics is that they produce declarative proof scripts. They are thus a step to narrow the gap between the declarative and the procedural style, which is still frequently used in practice.

We have implemented 15 declarative tactics, all of which come in a variant that produces a forward style proof as well as in a variant that produces a backward style proof. So far the experiments confirm our impression that declarative tactics are well suited to automate (sub)proofs having a common structure, as is the case for induction proofs or the integration of external systems such as computer algebra systems. Moreover, operations that depend on the syntactic structure of the formula can easily be expressed, for example, to provide structure for common forms of forward reasoning. In these situations, the declarative tactics were easy to write. However, for situations in which the subsequent proof steps are not known in advance, such as simplification, declarative tactics are not adequate.

theorem *th1:* $\lim_{x\to 3}\frac{x^2-5}{x-2}=4$
proof
  subgoals
    subgoal $|\frac{x^2-5}{x-2}-4|<\varepsilon$ **using** *A1:*$\varepsilon>0$ **and** *A2:*$|x-3|<?\delta$ **by** *factorbound*
    subgoal $?\delta>0$ **using** $\varepsilon>0$
  end by *limdefbw*
qed

---

theorem *th1:* $\lim_{x\to 3}\frac{x^2-5}{x-2}=4$
proof
  subgoals
    subgoal $|\frac{x^2-5}{x-2}-4|<\varepsilon$ **using** *A1:*$\varepsilon>0$ **and** *A2:*$|x-3|<?\delta$
    proof
      L1:$\frac{x^2-5}{x-2}-4=(x-3)*(\frac{1}{x-2})*(x-1)$ **by** *abeliandecide*
      $|x-1|\leq ?MV1$ **by** *linearbound*
      $|\frac{1}{x-2}|\leq ?MV2$ **by** *linearbound*
      L2: $|\frac{x^2-5}{x-2}-4|\leq|(x-3)*(\frac{1}{x-2})*(x-1)|$ **from** *L1*
      $.\leq|x-3|*|\frac{1}{x-2}|*|x-1|$
      $.<?\delta*?MV1*?MV2$
      $.\leq\varepsilon$
    qed
    subgoal $?\delta>0$ **using** $\varepsilon>0$
  end by *limdefbw*
qed

Figure 12: Declarative proof script of the example before and after processing the call of the declarative tactic `factorbound`

As already mentioned in the introduction, declarative proof languages and the verification of proof sketches has been studied by several people. There exists also several approaches to present a machine-found proof in a user friendly way [21,18]. In [27] a language is presented to automatically generate declarative proofs from proof terms. While this allows the presentation of proofs which have been found automatically, it does not deal with the specification of tactics in a declarative way.

Closely related to our work is ISAPLANNER [17]. ISAPLANNER generates proof plans and uses ISAR to represent them, that is, it also generates declarative proofs. It provides a "gap" command to represent open subgoals together with the annotation of a technique how to close such a gap. Compared to our approach, the main difference is that reasoning techniques are written as ML functions, whereas we use the underlying declarative proof language to specify the tactic. Moreover, our proof language differs from ISAR by allowing metavariables, which are not supported by ISAR, despite being supported by ISABELLE.

In the previous version of ΩMEGA, so-called *proof methods* were declaratively represented by *proof schemas*. Proof schemas were partial proofs in natural deduction (see [22]). In contrast to our approach, methods were implemented directly in the underlying programming language, no declarative proof language was used. Moreover, there was no possibility to pass control information in the form of a continuation.

Regarding intermediate tactic languages, our approach is similar to COQ's LTAC [12], which is an intermediate language intended to deal with small parts of proofs the user may like to automate locally. In contrast to our language, LTAC remains in the procedural style of the underlying tactic language instead of being declarative like our approach based on the declarative proof scripts. LTAC introduces conveniences of higher-level programming languages to the tactic script language which are independent form the underlying programming language and is similar in spirit to our aims. More specifically, LTAC provides pattern matching against the current goal, and our syntax for sequent patterns `|-` is inspired from it. LTAC also supports to match subterms and our syntax `[t]` is also the same here, except that we also allow to impose the polarity of the subformula supposed to match by `[t]+` or `[t]-`. A real extension of our language are the means to bind results of arbitrary computations to local script variables as well as the pattern syntax with ellipsis, which probably could be included in LTAC.

The matching part in case constructs of our tactic language is related to the extended meta-functions in ACL2 [23] which allow to access the current goal clause. The ACL2 meta-functions need to be proved correct in order to be usable by the ACL2 reasoner. From the LCF point of view, this is a way to include derived reasoning steps in the kernel proof rules, thus extending the kernel rules. In contrast to this our approach remains entirely in the LCF tradition since the declarative strategies generate proof scripts, which still need to be evaluated by the underlying (LCF-based) proof script interpreter. The possibility to perform arbitrary computations and bind the results to a term pattern, like the call to `maxima-factor` in the strategy *`factorbound`* is close to ACL2's `bind-free`, which takes an arbitrary binding list and adds it to the local context. This is also possible with our pattern approach by writing `X_1 .. X_N`, which has the advantage that the names of the local variables can be specified by the writer of the strategy. It would be possible to accommodate the `bind-free`-style in the pattern syntax, but so far we have not encountered situations where this was required. Moreover, the examples presented in [23] also bind only one variable.

In the context of rewriting several strategy languages exist. The general idea is to provide a language to specify a class of derivations the user is interested in by controlling the rule applications. Depending on the language, the language constructs are either defined by a combination of low-level primitives or build-in primitives. On a second layer, the languages provide constructs to express choice and sequencing, and recursion. Prominent examples are ELAN [7], MAUDE [25], and Stratego [31]. However, while being separate, these languages are not declarative in the sense that they are specified using a declarative language and produce declarative proofs.

References

1. Andreas Abel, Bor-Yuh Evan Chang, and Frank Pfenning. Human-readable machine-verifiable proofs for teaching constructive logic. In Uwe Egly, Armin Fiedler, Helmut Horacek, and Stephan Schmitt, editors, *Proceedings of the Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs (PTP'01)*. Universitá degli studi di Siena, June 2001.
2. Serge Autexier, Christoph Benzmüller, Dominik Dietrich, and Marc Wagner. Organisation, transformation, and propagation of mathematical knowledge in ΩMEGA. *Journal Mathematics in Computer Science*, 2(2):253–277, 2008.

3. Serge Autexier and Armin Fiedler. Textbook proofs meet formal logic - the problem of underspecification and granularity. In Michael Kohlhase, editor, *Mathematical Knowledge Management, 4th International Conference, MKM 2005, Revised Selected Papers*, volume 3863 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 2006.
4. Michael Beeson. Automatic generation of epsilon-delta proofs of continuity. In Jacques Calmet and Jan A. Plaza, editors, *Artificial Intelligence and Symbolic Computation, International Conference AISC'98, Plattsburgh, New York, USA, September 16-18, 1998, Proceedings*, volume 1476 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 1998.
5. Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors. *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*. Springer, 1999.
6. Woody W. Bledsoe. Challenge problems in elementary calculus. *J. Autom. Reasoning*, 6(3):341–359, 1990.
7. Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Christophe Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–95, 2001.
8. Robert S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.
9. Chad E. Brown. Verifying and invalidating textbook proofs using scunak. In Jonathan M. Borwein and William M. Farmer, editors, *Mathematical Knowledge Management, 5th International Conference, MKM 2006, Wokingham, UK, August 11-12, 2006, Proceedings*, volume 4108 of *Lecture Notes in Computer Science*, pages 110–123. Springer, 2006.
10. Jacques Calmet and Karsten Homann. Classification of communication and cooperation mechanisms for logical and symbolic computation systems. In *Frontiers of Combining Systems (FroCos)*, pages 221–234, 1996.
11. Pierre Corbineau. A declarative language for the Coq proof assistant. In Marino Miculan, Ivan Scagnetto, and Furio Honsell, editors, *Types for Proofs and Programs, International Conference, TYPES 2007, Cividale des Friuli, Italy, May 2-5, 2007, Revised Selected Papers*, volume 4941 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2007.
12. David Delahaye. A Proof Dedicated Meta-Language. *Electronic Notes in Theoretical Computer Science: Proceedings of Logical Frameworks and Meta-Languages (LFM), Copenhagen (Denmark)*, 70(2), July 2002.
13. Louise A. Dennis, Mateja Jamnik, and Martin Pollet. On the comparison of proof planning systems: λ-clam, ΩMEGA and IsaPlanner. *Electronic Notes in Theoretical Computer Science*, 151(1):93–110, 2006.
14. Dominik Dietrich and Ewaryst Schulz. Crystal: Integrating structured queries into a tactic language. *J. Autom. Reasoning*, 44(1-2):79–110, 2010.
15. Dominik Dietrich, Ewaryst Schulz, and Marc Wagner. Authoring verified documents by interactive proof construction and verification in text-editors. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *Intelligent Computer Mathematics, 9th International Conference, AISC 2008, 15th Symposium, Calculemus 2008, 7th International Conference, MKM 2008, Birmingham, UK, July 28 - August 1, 2008. Proceedings*, volume 5144 of *Lecture Notes in Computer Science*, pages 398–414. Springer, 2008.
16. Lucas Dixon and Jacques Fleuriot. A proof-centric approach to mathematical assistants. *J. of Applied Logic: Towards Computer Aided Mathematics Systems*, 4(4):505–532, 2005.
17. Lucas Dixon and Jacques D. Fleuriot. Isaplanner: A prototype proof planner in Isabelle. In Franz Baader, editor, *Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*, pages 279–283. Springer, 2003.

18. Armin Fiedler. *P.rex*: An interactive proof explainer. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning: Proceedings of IJCAR'01*, number 2083 in LNAI, pages 416–420, Siena, Italy, 2001. Springer.
19. Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF – A mechanised logic of computation*. Springer Verlag, 1979. LNCS 78.
20. John Harrison. Proof style. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs, International Workshop TYPES'96, Aussois, France, December 15-19, 1996, Selected Papers*, volume 1512 of *Lecture Notes in Computer Science*, pages 154–172. Springer, 1996.
21. Xiaorong Huang. *Human Oriented Proof Presentation: A Reconstructive Approach*. Number 112 in DISKI. Infix, Sankt Augustin, Germany, 1996.
22. Xiaorong Huang, Manfred Kerber, and Lassaad Cheikhrouhou. Adaptation of declaratively represented methods in proof planning. *Annals of Mathematics and Artificial Intelligence*, 23(3–4):299–320, 1998.
23. Warren A. Hunt Jr., Matt Kaufmann, Robert Bellarmine Krug, J S. Moore, and Eric Whitman Smith. Meta reasoning in acl2. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2005.
24. Daniel Kühlwein, Marcos Cramer, Peter Koepke, and Bernhard Schröder. The naproche system. In *Calculemus 2009 Emerging Trends*, pages 8–18, Grand Bent, Canada, July 2009.
25. Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Towards a strategy language for Maude. In Narciso Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications (WRLA 2004)*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 417–441. Elsevier, 2005.
26. Erica Melis and Jörg H. Siekmann. Knowledge-based proof planning. *Journal Artificial Intelligence*, 115(1):65–105, 1999.
27. Claudio Sacerdoti-Coen. Declarative representation of proof terms. *J. Autom. Reasoning*, 44(1-2):25–52, 2010.
28. Don Syme. Three tactic theorem proving. In Bertot et al. [5], pages 203–220.
29. A. Trybulec and H. Blair. Computer assisted reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th Int. Joint Conference on Artifical Intelligence*. M. Kaufmann, 1985.
30. Konstantin Verchinine, Alexander V. Lyaletski, and Andrey Paskevich. System for automated deduction (SAD): A tool for proof verification. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction*, volume 4603 of *Lecture Notes in Computer Science*, pages 398–403. Springer, July 2007.
31. Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA 2001)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.
32. Markus Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Bertot et al. [5], pages 167–184.
33. Freek Wiedijk. Formal proof sketches. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2004.
34. Vincent Zammit. On the implementation of an extensible declarative proof language. In Bertot et al. [5], pages 185–202.