

Recent Developments in Ω MEGA's Proof Search Programming Language

Serge Autexier Dominik Dietrich

German Research Centre for Artificial Intelligence Bremen (DFKI)

{serge.autexier|dominik.dietrich}@dfki.de

The development of interactive tactic based theorem provers started with the LCF system [10], 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 [18] or [11] 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 has led to *declarative proof languages*, inspired by MIZAR [16], 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 [17] 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 [5, 8, 15, 17]).

The paradigm of *proof planning* [7], which initially consisted of wrapping LCF-tactics into planning operators with pre- and post-conditions can be viewed as an attempt to exploit the macro-structure of proofs for the specification and organization of proof

search. Taking up this idea to integrate domain knowledge to heuristically guide proof search has been at the heart of the research activities of the Ω MEGA group since the beginning. It resulted in concepts to encode common patterns of reasoning, so-called *methods*, *control rules*, and *strategies*, as well as the integration of external specialist systems such as computer algebra systems, classical first order reasoner, or constraint solver. However, the formalisms to design methods, control rules and strategies where objects in Ω MEGA's underlying programming language, i.e. LISP.

In this paper we want to present the recent developments towards providing an *intermediate language* as an declarative, abstract layer to specify proof search knowledge, which is independent of the underlying programming language, and motivated by the following reasons: (i) A restricted language is easier to learn and therefore supports the user in writing his own tactics. (ii) Providing rich patterns allows the specification of complex tactics in a compact way. (iii) A separate language can easily be extended and allows the change of internals without breaking the functionality of proof search procedures. Moreover, it is a first step towards exchanging reasoning procedures between different proof assistants.

The starting point to have a well-defined separation line was to install *assertion-level* reasoning as the basic reasoning layer of Ω MEGA, which provided a suitable abstraction over the base logic. That layer consists of inference rules, which can either be derived from axioms or lemmas (collectively called *assertions*, see [2] and [3]) and thus are trusted, or else user-defined rules wrapping, for instance, calls to external reasoners, and thus untrusted. The first level of intermediate language concerns the declarative specification of inference rules as well as restrictions on how they are applied during proof search. This will be presented in Section 1. The next step was to provide a declarative language different from the underlying programming language to specify *strategies* (still more or less in the LCF-tactic style), which provide means to query inference rules from the context and provide search control constructs, and are briefly presented in Section 2. The last step so far was to provide a declarative proof language inspired by ISAR and especially to provide a declarative tactic language to specify strategies that generate declarative proof script, coming closer again to the ideas of operationalizing the proof structuring during proof. The tactic language is presented in Section 3.

A common characteristic of all three levels is to rely on compilation techniques, which result in more efficient code compared to their interpreted counterparts. Furthermore, optimizations can be performed independently of the language itself which increases robustness, and also allows for more efficiency when using domain and problem specific reasoning procedures. We strongly 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.

1. Annotated Inferences

Inference annotations allow the control of the search space explored in order to enable the application of inference rules, which are Ω MEGA's basic proof construction steps and which have the distinctive feature that they can be applied deeply, that is on proper subformulas of the formulas occurring in sequents. This allows for exponentially shorter proofs than in classical sequent calculus while increasing the number of possible applicable inference rules at each step. In practical applications, there is therefore a trade-off between the advantage of shorter proofs and the disadvantage that these short proofs may be harder to find, because there are more choice points in the search space. The inference annotation language allows the annotation of inference rules with search space restrictions. Annotated inferences are subsequently compiled into a primitive internal language. The benefits are twofold: (i) The annotations enable a high-level and fine-grained though declarative control of the internal search procedure, supporting the full range from top level inference to full deep inference as well as a mixture of them. (ii) Compilation results in reasonably efficient proof search procedures comparable to built-in procedures. Let us elaborate on these concepts more precisely.

Intuitively, an *inference* is a proof step with multiple premises and conclusions augmented by (1) a possibly empty set of hypotheses for each premise, (2) a set of *application conditions* that must be fulfilled upon inference application, (3) a set of *completion functions* that compute the values of premises and conclusions from values of other premises and conclusions, such as new meta-variables. Inferences are applied to tasks, which are a multi-conclusion sequents, expressing a subgoal to be proved. Initially there is only a single task consisting of the conjecture to be proved. A proof attempt is represented by an *agenda*, consisting of a set of tasks, a global substitution which instantiates meta-variables, and contextual information. Tasks are reduced to subtasks. In the case that the subtasks are empty, it is called *closed*, otherwise it is called *open*.

A preeminent feature of inferences is that they can be applied deeply inside formulas. Without going into details, let us note that polarities and uniform notation are sufficient to define a uniform notion of a *logical context* and to dynamically derive so-called replacement rules from that context (see [1] for details).

EXAMPLE 1.1. Consider the application of the axiom rule

$$\frac{P}{P} \text{ AXIOM}$$

and the following proof situation $\Gamma, (P^+ \Rightarrow^\beta Q^-)^- \vdash Q^+, \Delta$. Matching the conclusion of AXIOM to the goal Q^+ , and the premise against the Q^- within $(P^+ \Rightarrow^\beta Q^-)^-$ represents an inference match for the close direction of AXIOM, as both Q are α -related via \vdash . This allows us to close the goal Q^+ . However, the use of Q^- results in a new proof obligation, namely the goal P^+ , as it is β -related to Q^- . Consequently, we get the following transformation:

$$\Gamma, (P^+ \Rightarrow^\beta Q^-)^- \vdash Q^+, \Delta \xrightarrow{\text{AXIOM}} \Gamma, (P^+ \Rightarrow^\beta Q^-)^- \vdash P^+, \Delta$$

In order to enable the application of an inference, candidates for premises respectively the conclusion are searched in a sequent; by default, for a premise or conclusion we search inside all formulas of the sequent for subformulas that unify with the given formula. To control that search annotations within curly brackets can be attached to each premise and conclusion, respectively. Two variants of the axiom rule are shown below:

$$\frac{P}{P} \text{ axiom1} \qquad \frac{[P]\{*\vdash\}}{P} \text{ axiom2}$$

For *axiom1* the matching candidates of the premise are all top-level formulas on the left-hand side of the sequent, which is the default, whereas *axiom2* searches all subformulas on the left-hand side of the sequent, indicated by the square brackets followed by the pattern restricting the position. Thus only *axiom2* allows for the derivation

$$\frac{\Gamma, F \Rightarrow G \vdash F, \Delta}{\Gamma, F \Rightarrow G \vdash G, \Delta} \text{ axiom2}$$

We use the annotations $*$ and $-$ for a premise or conclusion to indicate that a partner must be found for it; in case of $*$ the matching formula is kept upon inference application while $-$ requires the matched formula to be replaced. For instance, the alternatively annotated inferences

$$\frac{[F]}{F \Rightarrow G\{*\}} \text{ impi1} \qquad \frac{[F]}{F \Rightarrow G\{-}} \text{ impi1}$$

both require to match the conclusion $F \Rightarrow G$, but only the second also requires to remove it upon application. As an effect they result respectively in the following derivations:

$$\frac{\Gamma, F \vdash G, F \Rightarrow G, \Delta}{\Gamma \vdash F \Rightarrow G, \Delta} \text{ impi1} \qquad \text{and} \qquad \frac{\Gamma, F \vdash G, \Delta}{\Gamma \vdash F \Rightarrow G, \Delta} \text{ impi2}$$

In order to identify a partner formula, higher-order unification is used in general. To restrict this for instance to HO-matching but also to more specific algorithms, such as for the application of inference rules, the keyword *check* allows to indicate a specific algorithm. Additional restrictions on positions can be expressed either by meta-predicates, or by more specific patterns on sequents.

The next class of annotations is to specify how the formula is traversed when looking for subformulas. By default the list of all candidate subformulas is returned. Alternatively one can use the annotations *outermost* or *innermost* in combination with either *leftmost* or *rightmost* that the search should stop on the first match; this can for instance be used to specify inferences to be used for a simplification using a leftmost, innermost strategy. Thus, the inference

$$\frac{[P]\{\text{leftmost, outermost}\}}{P} \text{ AXIOM}$$

prefers the formula F_1 over the formula F_2 in the proof situation $P \vee F_1, Q \wedge F_2 \vdash F_3$. However, the first occurrence of F introduces a proof obligation $\neg P$, while the second occurrence of F can be used without the introduction of new proof obligations. To restrict the applicability of the axiom rule to the latter, we provide the annotation *nopob*.

2. Declarative Language for Procedural Strategies

On top of inferences, proof assistance systems provide the layer of tactics to organize the proof search. These tactics are typically written in the underlying programming language and require the user to know about the internal data structures and functions. This not only hinders people not knowing these details from writing their own search procedures, but also makes third party tactic libraries not maintained by the proof assistance developers fragile with respect to changes of the underlying system in the sense that they may not compile anymore. To bridge the gap between the predefined proof operators and the programming language of the proof assistant we developed the tactic language **CRSIL** [9] as an intermediate abstraction layer. This new layer of abstraction can be seen in analogy to what has been done by introducing declarative proof languages.

```

strategy simplification
repeat
use (select lhs=rhs from current
  where (greaterlpo lhs rhs)) as forward
union
use (select lhs=rhs from current
  where (greaterlpo rhs lhs)) as backward

```

Figure 1. Selecting and annotating knowledge from current theory

It is clear that for complex and efficient proof strategies an expressive programming language is necessary: for that reason **CRStL** is extensible by user defined functions and predicates written in the programming language of the prover. However, small parts of proofs can often be automated by special purpose proof strategies, where a much simpler language suffices. It is desirable that a user can write such a strategy *on the fly* and requiring the use of the underlying programming language in such a case often prevents the non-expert user to take this option because he is unfamiliar with programming in the systems language. Even for experienced users it is often too time consuming to design a special purpose proof strategy in the underlying programming language when its use will be restricted only to a small part of a theory.

The language is arranged in two levels, a query language to access mathematical knowledge maintained in development graphs [12], and a strategy language to annotate the results of these queries with further control information. This comes from the insight that restricting the number of proof operators drastically reduces the search space and allows for a more efficient solution computation, provided that the filter is not too strict (see [14] and [13] for related work on relevance filtering). Therefore, the language explicitly supports the cycle select - process - search. Note that by the introduction of queries/filters tactics become dynamic objects that depend on the context. We call such adaptive tactics *theory-aware*. An example is shown in Figure 1, which shows a simplification strategy that selects all orientable equations from the current theory and applies them as long as possible.

The control layer emphasizes on providing language constructs to specify and integrate conditions in form of declarative patterns and meta-predicates on sequents, including the handling of subformulas and polarities. These conditions are used to specify applicability conditions of tactics, case analysis, or termination conditions for loops. Additionally, backtrack conditions can separately be installed to avoid the exploration of certain branches of the search space. An example is shown in Figure 2, which shows a simple tactic that performs a forward exploration to derive a subformula given as parameter of the tactic. Finally, Figure 3 gives an example in which the selected knowledge is further post processed using a Knuth-Bendix completion procedure.

```

strategy fwexplore
parameter formula
repeat
use select * from current.inferences as forward
until *, [formula]- |- *
  where (not (proofobligations (pos formula)))
backtrack-if (greater stratdepth 3)

```

Figure 2. Forward exploration with declarative termination condition

```

strategy KBGroup
use complete select lhs=rhs from groups as forward

```

Figure 3. Post-processing selected knowledge using Knuth Bendix completion

3. Declarative Language for Declarative Proof Script Strategies

The strategies from the previous section are still in the realm of LCF-tactics, which take a set of goals (agendas) and return a new, possibly empty set of goals. In [4] we presented a *declarative tactic language* on top of a *declarative proof language* (which can be seen as an extension of [9]). 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 differs from the language for procedural strategies in that the strategies are defined using an extended declarative proof script language by specifying intermediate proof states of the proof construction. These intermediate proof states act 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. The execution of a declarative tactic results in a declarative proof script, which in turn can be inserted into the document if desired. Thus, they provide a means to overcome the main problem of the declarative style of proof, namely that it is laborious to write and thus close the gap between both proof styles. Moreover, because of its additional abstraction, it might provide possibilities to exchange reasoning procedures between different proof assistants in the long-term view. Indeed, it has been noted in [17] that a declarative proof language can be implemented rather independently of the underlying logic.

As an example, we consider the problem of proving the theorem $\lim_{x \rightarrow 3} \frac{x^2-5}{x-2} = 4$. 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 5, where the declarative tactic `factorbound` (see Figure 4) 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: $\{LHS \mapsto x-3, RHS \mapsto ?\delta, GOALLHS \mapsto \frac{x^2-5}{x-2} - 4, GOALRHS \mapsto \varepsilon\}$
2. To be able to evaluate the **where** condition, the first **with** part is evaluated. This results in the following factorization: $Y_1 * \dots * Y_n = (x-3) * (\frac{1}{x-2}) * (x-1)$. Internally, a list $Y = [(x-3), (\frac{1}{x-2}), (x-1)]$ is generated and n is bound to 3. In the next assignment j is bound to 1 by looking up $x-3$ in the list of factors.
3. The conditions of the where part evaluate 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 5.

The example illustrates the following points: The declarative strategy is easy to specify as the overall problem can easily be divided in a sequence of subproblems. Due to the use of patterns, the strategy is independent of the number of factors computed by the computer algebra system. More importantly, it allows the formulation of what to do next after the factorization, namely to bound all but one factor.

4. Summary and Outlook

We have briefly reviewed the different concepts and language levels developed during our overall quest towards an declarative, abstract layer to specify proof search knowledge, which is independent of the underlying programming language of a proof assistant system.

```

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 4. Dynamic pattern matching and proof script generation

These are implemented in an experimental version of the Ω MEGA proof assistance system. Ongoing research is devoted to extend these ideas to the specification language layer and allow for the specification of tactics that generate or transform specifications, but also tactics that transform both specifications and proof scripts in combination.

References

- [1] S. Autexier. *Hierarchical Contextual Reasoning*. PhD thesis, Computer Science Department, Saarland University, Saarbrücken, Germany, 2003.
- [2] S. Autexier and D. Dietrich. Synthesizing proof planning methods and oants agents from mathematical knowledge. In J. M. Borwein and W. 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 94–109. Springer, 2006. ISBN 3-540-37104-4.
- [3] S. Autexier and D. Dietrich. Atomic metaduction. In B. Mertsching, editor, *Proceedings 32nd Annual German Conference on Artificial Intelligence*, LNCS, page 8, Paderborn, Germany, september 2009. Springer.
- [4] S. Autexier and D. Dietrich. A tactic language for declarative proofs. In M. Kaufmann and L. Paulson, editors, *International conference on Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 99–115, Edinburgh, Scotland, July 2010, to appear. Springer, Heidelberg.
- [5] S. Autexier and A. Fiedler. Textbook proofs meet formal logic - the problem of underspecification and granularity. In M. Kohlbase, 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. ISBN 3-540-31430-X.
- [6] Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin-Mohring, and L. 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*, 1999. Springer. ISBN 3-540-66463-7.

```

theorem th1: limx→3  $\frac{x^2-5}{x-2}$  = 4
proof
subgoals
  subgoal  $|\frac{x^2-5}{x-2} - 4| < \epsilon$  using A1: $\epsilon > 0$  and A2: $|x-3| < ?\delta$ 
    by factorbound
  subgoal  $?\delta > 0$  using  $\epsilon > 0$ 
  end by limdefbw
qed
-----
theorem th1: limx→3  $\frac{x^2-5}{x-2}$  = 4
proof
subgoals
  subgoal  $|\frac{x^2-5}{x-2} - 4| < \epsilon$  using A1: $\epsilon > 0$  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
      .<  $|x-3| * |\frac{1}{x-2}| * |x-1|$ 
      .<  $?\delta * ?MV1 * ?MV2$ 
      .<  $\epsilon$ 
    qed
  subgoal  $?\delta > 0$  using  $\epsilon > 0$ 
  end by limdefbw
qed

```

Figure 5. Declarative proof script of the example before and after processing the call of the declarative tactic `factorbound`

- [7] A. Bundy. The use of explicit plans to guide inductive proofs. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9th Conference on Automated Deduction*, number 310 in *LNCS*, pages 111–120, Argonne, Illinois, USA, 1988. Springer.
- [8] P. Corbineau. A declarative language for the Coq proof assistant. In M. Miculan, I. Scagnetto, and F. Honsell, editors, *Types for Proofs and Programs, International Conference, (TYPES 2007), Revised Selected Papers*, volume 4941 of *LNCS*, pages 69–84. Springer, 2007. ISBN 978-3-540-68084-0.
- [9] D. Dietrich and E. Schulz. Crystal: Integrating structured queries into a tactic language. *J. Autom. Reasoning*, 44(1-2):79–110, 2010.
- [10] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF – A mechanised logic of computation*. Springer Verlag, 1979. LNCS 78.
- [11] J. Harrison. Proof style. In E. Giménez and C. 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. ISBN 3-540-65137-3.
- [12] D. Hutter. Management of change in structured verification. In *Proceedings of Automated Software Engineering, ASE-2000*. IEEE, 2000.
- [13] J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009. doi: 10.1016/j.jal.2007.07.004.
- [14] W. Reif and G. Schellhorn. Theorem proving in large theories. In *In Proc. FTP 97*, pages 119–124. Kluwer Academic Publishers, 1998.
- [15] D. Syme. Three tactic theorem proving. In Bertot et al. [6], pages 203–220. ISBN 3-540-66463-7.
- [16] A. Trybulec and H. Blair. Computer assisted reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th Int. Joint Conference on Artificial Intelligence*. M. Kaufmann, 1985.
- [17] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Bertot et al. [6], pages 167–184. ISBN 3-540-66463-7.
- [18] V. Zammit. On the implementation of an extensible declarative proof language. In Bertot et al. [6], pages 185–202. ISBN 3-540-66463-7.