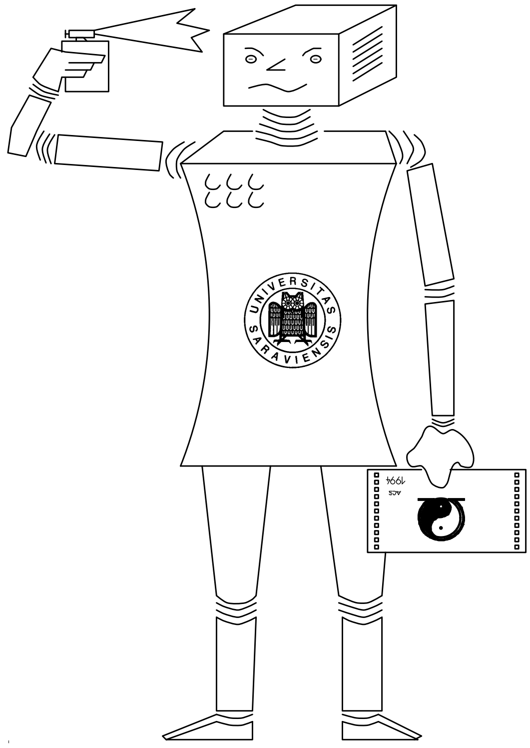
UNIVERSITÄT DES SAARLANDES
FACHBEREICH INFORMATIK
D-66041 SAARBRÜCKEN
GERMANY
WWW: http://js-sfbsun.cs.uni-sb.de/pub/www/



# Adapting Methods to Novel Tasks in Proof Planning

Xiaorong Huang, Manfred Kerber,
Michael Kohlhase, and Jörn Richts

**Published as:** In Bernhard Nebel and Leonie Dreschler-Fischer, editors, *KI-94: Advances in Artificial Intelligence*, Proceedings of the 18th German Annual Conference on Artificial Intelligence, pages 379–390, Saarbrücken, Germany, 1994. Springer Verlag, Berlin, Germany, LNAI 861.

# Adapting Methods to Novel Tasks in Proof Planning*

Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Jörn Richts

Fachbereich Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany
{huang|kerber|kohlhase|richts}@cs.uni-sb.de
WWW: http://js-sfbsun.cs.uni-sb.de/pub/www/

**Abstract.** In this paper we generalize the notion of method for proof planning. While we adopt the general structure of methods introduced by Alan Bundy, we make an essential advancement in that we strictly separate the declarative knowledge from the procedural knowledge. This change of paradigm not only leads to representations easier to understand, it also enables modeling the important activity of formulating meta-methods, that is, operators that adapt the declarative part of existing methods to suit novel situations. Thus this change of representation leads to a considerably strengthened planning mechanism.
After presenting our declarative approach towards methods we describe the basic proof planning process with these. Then we define the notion of meta-method, provide an overview of practical examples and illustrate how meta-methods can be integrated into the planning process.

## 1 Introduction

There has been growing concern in the automated theorem proving community that general purpose machine oriented procedures like resolution might have reached their limits in practice. Therefore the old discussion of the merits of a human-oriented vs. a machine oriented approach to automated theorem proving has been revived by researchers like Alan Bundy. In response to his request for a "science of reasoning" [3] a string of systems and theories that aim at combining human-oriented deduction methods with sophisticated planners have been proposed.

A central concept of knowledge based reasoning in mathematics and proof planning is that of a *method*. A method contains a piece of knowledge for solving or simplifying problems or transforming them into a form that is easier to solve. Therefore methods can be quite general such as finding proofs by a case analysis or complete induction, or the advice to expand definitions. On the other hand, domain specific methods are also very common, for instance, a clearly described proof sketch for proving a theorem by diagonalization or Bledsoe's $k$-parameter technique for proving the completeness of resolution calculi.

During his academic training a mathematician has to accumulate lots of methods. This body of methods is the reasoning repertoire which together with the factual knowledge, to a great extent forms his technical knowledge. Another

* This work was supported by the Deutsche Forschungsgemeinschaft, SFB 314 (D2)

equally important knowledge source of a mathematician is his ability to adapt existing methods to suit a new situation. Much of this discussion can already be found in George Pólya's analysis of mathematical reasoning "How to Solve It" [12], where he gives hundreds of examples for methods that mathematicians have to learn. Some of these have been stated very explicitly, others are very general and are largely illustrated with the help of examples only. Allen Newell [11] discussed the relevance of Pólya's heuristics very intensively, although he did not achieve a formalization. While specific methods have been widely implemented as so-called tactics in deduction systems like LCF [6] or Nuprl [4], and extended to methods in Alan Bundy's approach of proof planning [2], no adequate solution to the important problem of adapting methods to novel situations has been found so far. In this paper we propose an extension of Bundy's framework in order to attack precisely this question.

Alan Bundy views methods essentially as a triple consisting of a tactic, a precondition, and a postcondition. There the tactic is a piece of program code that can manipulate the actual proof in a controlled way. The precondition and the postcondition form a specification of the deductive ability of the tactic, formulating declaratively the applicability condition of the tactic and a description of the proof status after its application. This has been an essential progress compared with a mere tactic language because within this framework it is now possible to develop proof plans with the help of the declarative knowledge in the preconditions and postconditions. Following a one-sided approach relying on procedural knowledge only, the OYSTER-CLAM system developed by Bundy's group, still has however a severe drawback: the adaption of methods to other problems is almost impossible, because that would require the transformation of programs – tactics are just programs – which is known to be a very hard problem in practice.

To remedy this shortcoming our notion of method separates the procedural and the declarative knowledge in the tactic part.

## 2 General Framework

The work in this paper should be understood in the setting of a computational model that casts the entire process of theorem proving, from the analysis of a problem up to the completion of a proof, as an *interleaving process* of proof planning, method application and verification. In particular, this model ascribes a reasoner's reasoning competence to the existence of methods together with a planning mechanism that uses these methods for proof planning. Since only planning with methods and meta-methods accounts for the creative behavior of our approach, we will not elaborate on the verification phase in this paper.

The theorem proving process is centered around a partial proof tree, which accommodates concepts like *proof sketches*, *proof plans*, and *proofs* by providing various levels of certainty for justifications of proof nodes. Concretely the system follows the paradigm of a blackboard architecture, where all system components have access to the central data-structure of the blackboard, which in our case includes the current proof tree, the proof history, the method base, and a database of mathematical definitions, theorems, and proofs [8].

## 2.1 Methods

The concept of a method is central to the reasoning process, since methods are the basic units which make up proof plans and are carried out to obtain the proof by bridging the gap between premises and conclusions. The body of methods constitutes the basic reasoning repertoire of a reasoner, it is constantly adapted and enriched with increasing experience.

There was a long and heated debate in AI as to whether knowledge should be represented procedurally or declaratively. For both positions, arguments were put forward from psychological and computational perspectives with respect to, among others, flexibility, computational efficiency, and communicability. It has been realized that both forms of knowledge are necessary to simulate intelligent behavior. We believe it is plausible that both aspects play an important role in human theorem proving and in implementing human-oriented deduction systems. Therefore tactics in our methods have two parts, a declarative and a procedural one that interprets the declarative part. Usually in concrete methods the procedural part will be a standard procedure.

Generalizing the tactic part of a method from a procedure (in Bundy's framework) to a pair containing both a procedure and a piece of declarative knowledge is significant. By discerning the declarative part of tactics, it is now possible to formulate meta-methods that adapt the declarative part of existing methods and thus come up with novel methods. In a framework where a tactic only consists of procedural knowledge, we would in effect be confronted with the much more difficult problem of program synthesis. Our framework is cast so general that it accommodates both a small set of general purpose procedures which operate by applying pieces of domain-specific declarative knowledge, and an open-end set of special purpose reasoning procedures, in which knowledge needed is already implicitly incorporated.

We define a *method* to be a 6-tuple with the following components:

- *Rating*: A function evaluating the appropriateness of applying this method.
- *Premises*: A list of lines which are used to prove the conclusions.
- *Constraints*: Conditions that must hold before the method can be applied.
- *Conclusions*: A list of lines which are proved by this method.
- *Declarative content*: A piece of declarative knowledge used by the *rating* and the *procedural content*. We currently only deal with proof schemata.
- *Procedural content*: Either a standard procedure *interpreting* the piece of declarative knowledge, or a special purpose inference procedure.

Viewed within a planning framework, the method structure can be partitioned into different logic units as illustrated in figure 1. The *premises*, the *constraints*, and the *conclusions* slots together specify whether a method is applicable in a particular proof state. The *rating* is a procedure that evaluates the declarative content of the method, the proof history, and the planning state to give an estimation of the probability of a significant contribution of this method. Thus the rating is a central part of the heuristic mechanism that guides the search for plans (for an example of a rating-driven proof planning system see [13]).

While the specification contains all necessary information for the planner, the *declarative content* and the *procedural content* slots *together* play the role of

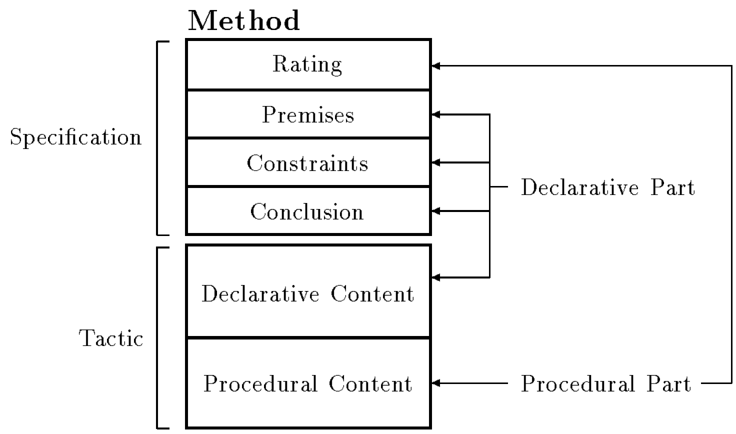



**Fig. 1.** The Structure of Methods

a so-called *tactic* which is a pure procedure in systems like Nuprl [4] or Bundy's framework for proof planning [3]. For the purposes of this paper it suffices to think of the declarative content as a proof schema with meta-variables, and of the procedural content as a LISP, C, or PROLOG procedure that takes this schema as an argument. The possibilities range from the case where the procedure is an interpreter that matches the proof schema and inserts it into the current proof tree with the meta-variables bound to a situation where the procedure completely ignores the proof schema and constructs new lines purely procedurally. The latter subcase is what can be found in [2]. We only insist that the value of the procedure is a subproof tree that can be integrated into the current partial proof tree.

### 2.2 Proof Planning with Methods

To give an account of the proof planning process itself, we first remember that the goal of proof planning is to fill gaps in a given partial proof tree by forward and backward reasoning. In our framework we follow a STRIPS-like planning paradigm [5], where the planning operators correspond to the methods. Thus from an abstract point of view the planning process is the process of exploring the search space of *planning states* that is generated by the *planning operators* in order to find a complete *plan* (that is a sequence of instantiated planning operators) from a given *initial state* to a *terminal state*.

Concretely a *planning state* is a subset of lines in the current partial proof that correspond to the boundaries of a gap in the proof. This subset can be divided into *open lines* (that must be proved to bridge the gap) and *support lines* (that can be used to bridge it). The *terminal state* is reached when there are no more open lines. In the planning process new open lines enter the planning state as subgoals by backward reasoning from existing open lines and new support lines by forward reasoning from existing support lines. In order to achieve this with a uni-directional planning mechanism, the planning direction must be independent of the reasoning direction.

The key feature of our approach is that the planning operators are directly derived from the specifications of the methods. However, the specification only

| | | |
|---|---|---|
| **Method** `def-i` (Definition Introduction) | | |
| **Premises** | `Definition:` | $\forall \overline{x} \centerdot P(\overline{x}) \Leftrightarrow \Psi_{\overline{x}}$ |
| | $\oplus$ `expanded-line:` | $\Psi_{\overline{t}}$ |
| **Conclusions** | $\ominus$ `defined-line:` | $P(\overline{t})$ |

| | | |
|---|---|---|
| **Method** `def-e` (Definition Elimination) | | |
| **Premises** | `Definition:` | $\forall \overline{x} \centerdot P(\overline{x}) \Leftrightarrow \Psi_{\overline{x}}$ |
| | $\ominus$ `defined-line:` | $P(\overline{t})$ |
| **Conclusions** | $\oplus$ `expanded-line:` | $\Psi_{\overline{t}}$ |

**Fig. 2.** The specification of the methods `def-i` and `def-e`

gives a static description (viewed from the completed proof) of the method which is inadequate for the dynamic behavior needed in proof planning. Statically a method derives its *conclusions* from its *premises*. Dynamically, it is important to declare which lines in the specification have to be present in the planning state for the method to be applicable (we will call them *required* lines), and which are constructed by the method. We will do this by labeling the latter lines which will be inserted into the planning state by the label "$\oplus$". Additionally it is useful to specify that some of the required lines will not be used again by the planner. We will mark such lines with a "$\ominus$". Note that the required lines consist of the unmarked ones and those that are marked with "$\ominus$". This labeling in effect determines the direction (forward vs. backward) of reasoning of the method. In order to illustrate these labels, figure 2 shows the specification of two simple methods. These methods are only a simplified version of a more general class of methods applying assertions (definitions and theorems). For a study of this class which approximates basic proof steps encountered in informal mathematical practice see [7].

Planning State 1

| | | | | |
|---|---|---|---|---|
| (!) | 1. | 1; | $\vdash$ | $\forall \sigma \centerdot \mathsf{symmetric}(\sigma) \Leftrightarrow \forall x, y \centerdot \langle x, y\rangle \in \sigma \Rightarrow \langle y, x\rangle \in \sigma$ |
| (!) | 2. | 2; | $\vdash$ | $\mathsf{symmetric}(\rho)$ |
| (?) | 9. | 1,2; | $\vdash$ | $\mathsf{symmetric}(\mathsf{converse}(\rho))$ |

`def-i` 1 9
`def-e` 1 2

Planning State 2

| | | | | |
|---|---|---|---|---|
| (!) | 1. | 1; | $\vdash$ | $\forall \sigma \centerdot \mathsf{symmetric}(\sigma) \Leftrightarrow \forall x, y \centerdot \langle x, y\rangle \in \sigma \Rightarrow \langle y, x\rangle \in \sigma$ |
| (!) | 3. | 1,2; | $\vdash$ | $\forall x, y \centerdot \langle x, y\rangle \in \rho \Rightarrow \langle y, x\rangle \in \rho$ |
| (?) | 8. | 1,2; | $\vdash$ | $\forall x, y \centerdot \langle x, y\rangle \in \mathsf{converse}(\rho) \Rightarrow \langle y, x\rangle \in \mathsf{converse}(\rho)$ |

**Fig. 3.** Using `def-i` and `def-e` in the planning process

Figure 3 shows an example of effect of the methods **`def-e`** and **`def-i`** on the planning state. In planning states we mark open lines by "?" and support lines by "!". The method **`def-i`** applies a definition for the predicate $P$ to an open line and **`def-e`** applies it to a supporting line. It is obvious that in both methods the line of the definition is required when applying these methods because it is not sensible to "guess" a definition[2]; furthermore this line must not be deleted since it might be used more than once. Therefore `Definition` has no label. Clearly in both methods `defined-line` must be a required line and `expanded-line` can be constructed by the methods (and therefore is labeled with "$\oplus$"). Furthermore `defined-line` in `def-i` (and analogously in `def-e`) is useless after the application because an open line must be proved only once.

In figure 4 we give an abstract view of the planning algorithm. In this algorithm matching must consider the distribution of labels in order to apply the planning operators in the correct direction. In particular an open line in the planning state can only be matched against a line from the *conclusions* slot; similarly support lines can only be matched against *premises*. During the matching of the required lines (in steps 1.(b) and 1.(c)) and the evaluation of the constraints (step 1.(d)) all meta-variables should have been bound to terms without meta-variables (in fact we consider this as one of the applicability conditions). Therefore the new lines of step 4.(a) can be constructed by simply instantiating the meta-variables. The new lines in the *premises* slot are inserted as open lines while the lines from the *conclusions* slot become support lines.

1. Find all applicable methods
   (a) Select a line $L$ from the planning state.
   (b) Select a method $M$ with a required line that *matches* $L$.
   (c) *Match* every required line in $M$ with a line in the planning state.
   (d) Evaluate the *constraints* of $M$.
2. Calculate the ratings of Methods
3. Select the best method $\widetilde{M}$ (this is the choice point for backtracking)
4. Apply the planning operator to the planning state
   (a) *Insert* the new lines *constructed* by $\widetilde{M}$.
   (b) Delete the lines marked with a "$\ominus$" in $\widetilde{M}$ from the planning state.

**Fig. 4.** The planning algorithm

Once a complete proof plan is found, all methods in the proof plan are successively applied. Note that such a method application phase need not lead to a complete proof of the problem at hand, since we do not require methods to be sound or complete with respect to their specifications. Furthermore the proof segments inserted by the methods may still contain open lines (see e.g. the `hom1-1` method) that define further gaps that still have to be closed by the proof planner. Therefore the verification phase may result in a recursive call to the planner or in backtracking. While the first possibility calls for a refinement of the plan found and can be used to model hierarchical planning, the latter rejects the plan and calls the proof planner in order to find a different plan.

Now that we have understood the basic framework, we have a look at the slightly more complex, related example of the `hom1-1` method (see figure 5). Its proof strategy can informally be described by: *If $f$ is a given function, $P$ a defined predicate and the goal is to prove $P(f(c))$, then show $P(c)$ and use this*

[2] This could be sensible at a more sophisticated level of proof planning. However, this "guessing" should be implemented by a different method in order to clearly separate it from simply applying a definition.

*to show* $P(f(c))$. The very idea is that $f$ is a homomorphism for the property $P$ and that $f$ can be "rippled out" (compare [2, 9]). Note that line 5 is an open line that does not occur in the specification and therefore does not enter the planning state. This leads to an abstraction in the planning process (i.e. there is less information in the planning state): since line 5 is not considered by the planner, after completing the plan it will be inserted into the proof tree as an open line by the application of the tactic of `hom1-1`. This will result in a recursive call of the planner in the following verification phase.

| Method: `hom1-1` | | | | | |
|---|---|---|---|---|---|
| rating | `rating-hom1-1` | | | | |
| prem | 1, 2, $\oplus$3 | | | | |
| constr | — | | | | |
| conc | $\ominus$6 | | | | |
| dec-cont | 1. | 1; | $\vdash$ | $\forall x_\blacksquare \text{Formula}_f$ | (J1) |
| | 2. | 2; | $\vdash$ | $\forall x_\blacksquare P(x) \Leftrightarrow \Psi_x$ | (J2) |
| | 3. | 1,2; | $\vdash$ | $P(c)$ | (J3) |
| | 4. | 1,2; | $\vdash$ | $\Psi_c$ | (`def-e` 2 3) |
| | 5. | 1,2; | $\vdash$ | $\Psi_{f(c)}$ | (OPEN 1 4) |
| | 6. | 1,2; | $\vdash$ | $P(f(c))$ | (`def-i` 2 5) |
| proc | `schema-interpreter` | | | | |

**Fig. 5.** The `hom1-1` method

For example, to prove that the converse relation of a binary relation $\rho$ is symmetric (formally: $\mathsf{symmetric}(\mathsf{converse}(\rho))$), the method `hom1-1` can be applied by substituting $\mathsf{converse}$, $\mathsf{symmetric}$, and $\rho$ for the meta-variables $f$, $P$, and $c$, respectively. While in figure 3 we filled the gap between $\mathsf{symmetric}(\rho)$ and $\mathsf{symmetric}(\mathsf{converse}(\rho))$ which were both existing lines, in this example the method `hom1-1` proposes $\mathsf{symmetric}(\rho)$ as a new line which can be used to prove $\mathsf{symmetric}(\mathsf{converse}(\rho))$ together with the definitions of $\mathsf{symmetric}$ and $\mathsf{converse}$. If the plan can be completed (by proving the new open line 3 with some additional information about $\rho$) the proof resulting from the application of the tactic `hom1-1` would look like in figure 6.

| | | | | |
|---|---|---|---|---|
| 1. | 1; | $\vdash$ | $\forall \sigma_\blacksquare \forall x, y_\blacksquare \langle x, y\rangle \in \mathsf{converse}(\sigma) \Leftrightarrow \langle y, x\rangle \in \sigma$ | (J1) |
| 2. | 2; | $\vdash$ | $\forall \sigma_\blacksquare \mathsf{symmetric}(\sigma) \Leftrightarrow \forall x, y_\blacksquare \langle x, y\rangle \in \sigma \Rightarrow \langle y, x\rangle \in \sigma$ | (J2) |
| 3. | 1,2; | $\vdash$ | $\mathsf{symmetric}(\rho)$ | (J3) |
| 4. | 1,2; | $\vdash$ | $\forall x, y_\blacksquare \langle x, y\rangle \in \rho \Rightarrow \langle y, x\rangle \in \rho$ | (`def-e` 3 2) |
| 5. | 1,2; | $\vdash$ | $\forall x, y_\blacksquare \langle x, y\rangle \in \mathsf{converse}(\rho) \Rightarrow \langle y, x\rangle \in \mathsf{converse}(\rho)$ | (OPEN 1 4) |
| 6. | 1,2; | $\vdash$ | $\mathsf{symmetric}(\mathsf{converse}(\rho))$ | (`def-i` 2 5) |

**Fig. 6.** The proof resulting from the application of the tactic `hom1-1`

Let us have a look at the justifications in this proof fragment. Justifications J1 and J2 are found by the matching procedure when applying the planning operator of `hom1-1`, J3 will be inserted by the remaining proof plan. The justifications of lines 4 and 6 stand for the subproofs generated by the applications of the tactics of these methods, whereas the justification of line 5 defines a new gap with support lines containing lines 1 and 4.

## 3 Extending the Reasoning Repertoire by Meta-Methods

It is one of the main features contributing to the problem solving competence of mathematicians that they can extend their current problem solving repertoire by adapting existing methods to suit novel situations (see [12] for mathematical reasoning and [14] for general problem solving). By adopting a declarative approach for formulating methods, we propose a way to mechanize some aspects of this procedure. In the rest of this paper, the emphasis is laid on the notion of meta-method and a description of classes of meta-methods, while the integration of meta-methods in the planning process is largely left to further research.

### 3.1 Definition of Meta-Methods

The isolation of the declarative part of tactics makes it feasible to formulate meta-methods adapting existing methods. To achieve the same effect, in a framework where tactics consist only of procedural knowledge, we would be confronted with the much more difficult problem of adapting procedures.

A *meta-method* consists of:[3]

- A *body*: a procedure which takes as input a method, and possibly further parameters (in particular the current state of proof planning) and generates a new method with the same procedural part (see figure 1).
- A *rating*: a procedure which takes as input a method, the current state of proof planning and the proof history. It estimates the contribution of the application of the meta-methods to the solution of the current problem.

We illustrate this definition with the help of the `hom1-1` method in figure 5. This method simplifies a problem by generating an intermediate goal, where a *unary* function symbol is eliminated. Suppose we are facing the similar problem of proving that the intersection of symmetric relations is itself a symmetric relation. A variant of `hom1-1` is needed, which handles a *binary* function symbol (i.e. "$\cap$") in a similar way.

| | |
|---|---|
| **Meta-Method** | `add-argument(M,F)` |
| **Rating** | `add-argument-rating` |
| **Procedure** | `add-argument-proc` |

**Fig. 7.** Meta-Method: `add-argument`

In the following we illustrate how the meta-method `add-argument` (figure 7) generates a binary version `hom1-2` (figure 8) from the unary version `hom1-1` (figure 5). While `hom1-1` is applicable to situations with a unary predicate constant $P$ and a unary function constant $f$, `hom1-2` handles situations with a unary

[3] In general, a more complete specification will be necessary for more complex problems where meta-level planning is necessary for the generation of a new method.

| Method: `hom1-2` | |
|---|---|
| rating | `rating-hom1-1` |
| prem | 1, 2, $\oplus 3$, $\oplus 4$ |
| constr | — |
| conc | $\ominus 8$ |
| dec-cont | 1. 1; $\vdash$ $\forall x, y_\blacksquare \text{Formula}_g$ (J1)<br>2. 2; $\vdash$ $\forall x_\blacksquare P(x) \Leftrightarrow \Psi_x$ (J2)<br>3. 1,2; $\vdash$ $P(c)$ (J3)<br>4. 1,2; $\vdash$ $P(d)$ (J4)<br>5. 1,2; $\vdash$ $\Psi_c$ (`def-e` 2 3)<br>6. 1,2; $\vdash$ $\Psi_d$ (`def-e` 2 4)<br>7. 1,2; $\vdash$ $\Psi_{g(c,d)}$ (OPEN 1 5 6)<br>8. 1,2; $\vdash$ $P(g(c,d))$ (`def-i` 2 7) |
| proc | `schema-interpreter` |

**Fig. 8.** The `hom1-2` method

predicate constant $P$ and a binary function constant $g$. Note that $P$, $f$, and $g$ are meta-variables standing for object constants.

The meta-method `add-argument` takes as input a method `M` and a unary function or predicate constant `F` used in `M`. This meta-method is supposed to add an argument to the key constant symbol `F`, the modified constant is called `F'`. The rating procedure yields the value zero if `F` does not occur in `M`, an average value if `F` does not occur in the premises and conclusions of `M`. It produces a high value if `F` is a key symbol of the method `M`.

The procedure `add-argument-proc` creates a method `M'` by carrying out the following modification on the declarative part of `M`:

- replace $\texttt{F}(x)$ by $\texttt{F}'(x, y)$ and augment the corresponding quantifications,
- replace $\texttt{F}(a)$ by $\texttt{F}'(a, b)$ ($b$ has to be a new meta-variable),
- if $a$ occurs in a proof line, but not in a term $\texttt{F}(a)$, a copy of this line will be inserted into the proof schema, replacing $a$ by $b$ (in the example line 4 is copied from 3).

Let us reiterate the crucial advantage of separating the procedural and the declarative knowledge: the procedural part of `M` can be taken over for the new method.

| | | | | |
|---|---|---|---|---|
| 1. | 1; | $\vdash$ | $\forall \rho, \sigma_\blacksquare \forall x, y_\blacksquare \langle x, y\rangle \in (\rho \cap \sigma) \Leftrightarrow \langle x, y\rangle \in \rho \wedge \langle x, y\rangle \in \sigma$ | (J1) |
| 2. | 2; | $\vdash$ | $\forall \sigma_\blacksquare \mathsf{symmetric}(\sigma) \Leftrightarrow \forall x, y_\blacksquare \langle x, y\rangle \in \sigma \Rightarrow \langle y, x\rangle \in \sigma$ | (J2) |
| 3. | 1,2; | $\vdash$ | $\mathsf{symmetric}(\rho)$ | (J3) |
| 4. | 1,2; | $\vdash$ | $\mathsf{symmetric}(\sigma)$ | (J4) |
| 5. | 1,2; | $\vdash$ | $\forall x, y_\blacksquare \langle x, y\rangle \in \rho \Rightarrow \langle y, x\rangle \in \rho$ | (`def-e` 2 3) |
| 6. | 1,2; | $\vdash$ | $\forall x, y_\blacksquare \langle x, y\rangle \in \sigma \Rightarrow \langle y, x\rangle \in \sigma$ | (`def-e` 2 4) |
| 7. | 1,2; | $\vdash$ | $\forall x, y_\blacksquare \langle x, y\rangle \in (\rho \cap \sigma) \Rightarrow \langle y, x\rangle \in (\rho \cap \sigma)$ | (OPEN 1 5 6) |
| 8. | 1,2; | $\vdash$ | $\mathsf{symmetric}((\rho \cap \sigma))$ | (`def-i` 2 7) |

**Fig. 9.** The proof resulting from the application of the tactic `hom1-2`

In figure 9 it is shown how the `hom1-2` method simplifies the problem of showing that the intersection of two symmetric relations is symmetric too. Analogously a method `hom2-1` (for handling a unary function symbol and a binary predicate symbol) can be obtained by applying `add-argument` with the arguments `hom1-1` and $P$.

### 3.2 Proof Planning with Meta-Methods

Meta-methods can be incorporated into the planning algorithm of figure 4. To do this, firstly it must be possible to interrupt the planning with methods, in order to create a new method with meta-methods. In our approach this is done when all applicable methods yield a rating below a certain threshold. The harder question is the choice of a meta-method and a method for the current situation. We believe that there can hardly be any general answer and we have to rely on heuristics. In an interactive proof development environment like Ω-MKRP [8] the user might want to make this choice himself. Therefore our main emphasis lies in the task of offering the user heuristic support for this choice. Even more challenging would be an automation, of course. A trivial answer would be to apply all existing meta-methods on all existing methods and then choose the applicable one with the highest rating. Such a procedure can be fairly expensive with a large knowledge base. The first heuristics for choosing a method to adapt we will investigate are listed below:

- Organize methods in a hierarchy of mathematical theories and prefer methods that belong to the same theory as the current problem or whose theory is close to that of the problem in the hierarchy.
- Use general conflict solving strategies like those of OPS5 [1], for instance, favor the methods and meta-methods with the most specific specification.
- Take only non-applicable methods with the highest rating[4].

Naturally only successful methods generated in a short-term memory are integrated into the permanent base of methods. Another way to reduce the cost of the operation would be to create only the specification of the methods to be generated, select one for application and create the tactic part by need.

### 3.3 Classes of Meta-Methods

Clearly the success of the approach outlined in this paper critically depends on the body of meta-methods that is at the disposal of the planner. In order to get an idea of the range of possible meta-methods, consider the following classes.

**Generalization** This type of meta-methods is designed to generalize an existing method to extend the class of problems it can solve.

  - Precondition analysis: a new method may be produced by generalizing or even removing some of the lines in the specification.
  - Syntactic abstraction: this type of meta-methods abstracts a given method by replacing terms satisfying certain conditions with corresponding meta-variables.

[4] This presupposes sophisticated rating functions, which yield meaningful ratings even if the method in question is not applicable.

- Semantic abstraction: this type of meta-methods produces more abstract methods by capturing the nature of an existing method. For example, a meta-method `fixpoint-freeness` can abstract a concrete method solving the Cantor problem. It recognizes that the very idea of the method is based on the fixpoint-freeness of a function.

**Syntactical adaption** At least two types of gaps require syntactical adaptions of methods. In the first case, the problem to be solved is basically the same as the one a method is designed to solve, but the original is not applicable due to a different formulation. In the second case, some non-trivial syntactical adaption is necessary in order to solve a related but different problem.

- Change of formulation: Many mathematical concepts are logically identical, in certain sense. For example, sets can be viewed as predicates, and predicates as special functions that yield only boolean values.
  - `set-to-predicate` generates methods by either replacing formulations concerning sets by formulations concerning predicates, or in the opposite direction.
  - `predicate-to-function` and `set-to-function` are similar to `set-to-predicate`.
- Change of syntactical structure:
  - `add-argument` (illustrated in detail in this section).
  - `connective-to-quantifier` generating a method handling problems containing universal quantifiers from a methods containing the logic connective $\wedge$.

For more high-level procedures like learning and analogical reasoning further classes of meta-methods are necessary, for instance, a meta-method that creates new methods by isolating interesting parts in a method.

## 4 Conclusion

The advanced problem solving competence of a mathematician relies mainly on his ability to adapt problem solving knowledge to new situations where existing methods are not directly applicable. Up to now this has not received enough attention in the field of automated theorem proving. In this paper we have proposed a refined proof planning approach in order to mechanize parts of this ability. It is our conviction that in the proof planning framework this is only possible with declaratively represented knowledge. Therefore we have defined a new structure for representing methods which supports the separation of the knowledge into a declarative and a procedural part.

Since methods describe the planning operators (represented in the "specification") as well as the deduction procedures (represented in the "tactic"), the separation of the declarative and the procedural knowledge is necessary in the specification and in the tactic as well. In order to perform the adaption of methods, we have introduced the new notion of a meta-method. We have specified a proof planning process based on methods and meta-methods. The approach has not only proved to be useful on the investigated examples, but also offers the possibility to formalize analogical reasoning [10] and the basis for learning of methods. Currently we are implementing these ideas in order to gain new experiences from the experiments and to concentrate on the important point of heuristic control through the ratings.

**Acknowledgments**

We would like to thank Jörg Denzinger, Erica Melis, Arthur Sehn, and Inger Sonntag for many fruitful discussions about proof plans, which inspired and clarified many of the ideas presented here.

## References

1. L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5 – An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, Massachusetts, USA, 1985.
2. A. Bundy. The use of explicit plans to guide inductive proofs. In *Proc. of 9th International Conference on Automated Deduction*, pages 111–120. Springer, 1988.
3. A. Bundy. A science of reasoning: Extended abstract. In *Proc. of 10th International Conference on Automated Deduction*, pages 633–640. Springer, 1990.
4. R. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, New Jersey, 1986.
5. R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, **2**:189–208, 1971.
6. M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation.* LNCS 78. Springer, 1979.
7. X. Huang. Reconstructing proofs at the assertion level. In *Proc. of 12th International Conference on Automated Deduction.* Springer, 1994.
8. X. Huang, M. Kerber, M. Kohlhase, E. Melis, D. Nesmith, J. Richts, and J. Siekmann. Ω-MKRP – a proof development environment. In *Proc. of 12th International Conference on Automated Deduction.* Springer, 1994.
9. D. Hutter. Guiding induction proofs. In *Proc. of the 10th International Conference on Automated Deduction*, pages 147–161. Springer, 1990.
10. E. Melis. Change of representation in theorem proving by analogy. SEKI-Report SR-93-07, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, 1993.
11. A. Newell. The heuristic of George Polya and its relation to artificial intelligence. In R. Groner, M. Groner and W. F. Bishof, editors, Methods of Heuristics, Lawrence Erlbaum, Hillsdale, New Jersey, USA, 195-243.
12. G. Pólya. *How to Solve it.* Princeton Univ. Press, 1945.
13. I. Sonntag and J. Denzinger. Extending automatic theorem proving by planning. SEKI-Report SR-93-02 (SFB), Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, Germany, 1993.
14. K. VanLehn. Problem solving and cognitive skill acquisition. In M. I. Posner, editor, *Foundations of Cognitive Science*, chapter 14. MIT Press, 1989.