UNIVERSITÄT DES SAARLANDES
FACHBEREICH INFORMATIK
D-66041 SAARBRÜCKEN
GERMANY

WWW: http://js-sfbsun.cs.uni-sb.de/pub/www/

SEKI Report

Planning Mathematical Proofs with Methods

Xiaorong Huang Manfred Kerber
Jörn Richts Arthur Sehn

SEKI Report SR-94-08

Planning Mathematical Proofs with Methods

Xiaorong Huang Manfred Kerber Jörn Richts Arthur Sehn
Fachbereich Informatik, Universität des Saarlandes
Postfach 15 11 50, D-66041 Saarbrücken, Germany
{huang|kerber|richts|acsehn}@cs.uni-sb.de

Abstract

In this report we formally describe a declarative approach for encoding plan operators in proof planning, the so-called methods. The notion of method evolves from the much studied concept tactic and was first used by A. Bundy. Significant deductive power has been achieved with the planning approach towards automated deduction, however, the procedural character of the tactic part of methods hinders mechanical modification. Although the strength of a proof planning system largely depends on powerful general procedures which solve a large class of problems, mechanical or even automated modification of methods is nevertheless necessary for at least two reasons. Firstly methods designed for a specific type of problems will never be general enough. For instance, it is very difficult to encode a general method which solves all problems a human mathematician might intuitively consider as a case of homomorphy. Secondly the cognitive ability of adapting existing methods to suit novel situations is a fundamental part of human mathematical competence. We believe it is extremely valuable to computationally account for this kind of reasoning.

The main part of this report is devoted to a declarative language for encoding methods, composed of a tactic and a specification. The major feature of our approach is that the tactic part of a method is split into a declarative and a procedural part in order to enable a tractable adaption of methods. The applicability of a method in a planning situation is formulated in the specification, essentially consisting of an object level formula schema and a meta-level formula of a declarative constraint language. After setting up our general framework, we mainly concentrate on this constraint language. Furthermore we illustrate how our methods can be used in a STRIPS-like planning framework. Finally we briefly show the mechanical modification of declaratively encoded methods. An annotated runtime protocol of an example can be found in the appendix.

Chapter 1

Introduction

> So fängt denn alle menschliche Erkenntnis mit Anschauungen an, geht von da zu Begriffen und endigt mit Ideen.
>
> *Immanuel Kant, Kritik der reinen Vernunft*

Mathematicians learn during their academic training not only facts like definitions or theorems, but also problem-solving *know-how* for proving mathematical theorems. An important part of this know-how can be described in terms of reasoning methods like the diagonalization procedure, the application of a definition, or the application of the homomorphy property. The main aim of this report is to formalize the concept of a *method* in order to reflect more closely the meaning of the notion method of the informal mathematical language. The importance of plausible reasoning in proof search has been pointed out by Pólya [24, p. vi]: "*There are two kinds of reasoning, as we said: demonstrative reasoning and plausible reasoning. ... In strict reasoning the principal thing is to distinguish a proof from a guess, a valid demonstration from an invalid attempt. In plausible reasoning the principal thing is to distinguish a guess from a guess, a more reasonable guess form a less reasonable guess. ... [plausible reasoning] is the kind of reasoning on which his [a mathematician's] creative work will depend*". In this report we concentrate ourselves exactly on the creative part. We try to capture some aspects of plausible reasoning by a proof planning process built on top of the methods as plan operators.

The notion *method* also stands in a long tradition in human-oriented mechanical theorem proving. One of the most advanced early proof checking systems is de Bruijn's Automath system [4]. The main motivation for this system continues to serve as a guideline for nowadays systems: only a small part of mathematical literature today is absolutely flawless. To improve this situation, an interactive proof checker should be used in order to carry out the meticulous final checking. Automath offers to that purpose an interactive environment, where the user must build a proof at a low calculus level. However, proving theorems at the logical level is very cumbersome. In order to overcome this, more recent systems like Nuprl [9], Isabelle [22], and IMPS [10] employ a mechanism called *tactic*. Tactics are programs that enable the user to comfortably manipulate the current proof state. One user interaction, namely the call of a single tactic, results in a sequence of calculus-level inference steps. While these systems are finding increasing acceptance and have also been put into use with remarkable success, there is one major objection nevertheless: They

incorporate little automated deductive support beyond the tactic level.

In order to provide such an automatic support, Bundy et al. developed in the OYSTER-CLAM-system [7] a plan-based approach, where the tactics are extended to methods. A *method* can be viewed as a unit consisting of a procedural tactic *and* a declarative specification. The latter allows reasoning *with* methods and in particular enables the use of methods as *plan operators*. These techniques have been applied in particular to problems based on mathematical induction. For example, a large part of the heuristic knowledge of the Boyer-Moore prover [3] has been encoded into such methods [5].

To give an idea of the notion method consider the *homomorphy* methods. For instance, it can be useful to prove the symmetry of the intersection of two binary relations by showing that each of the two relations is symmetric. That means in order to show $symmetric(\rho \cap \sigma)$ it might be a good idea to show $symmetric(\rho)$ and $symmetric(\sigma)$ separately and then to combine these proofs to a proof of the symmetry of the intersection. The `hom1-2` method, which will be explained in detail in chapter 5, suggests just this proof idea.

A homomorphism can occur in very different syntactic shapes. Another version may involve a unary function (e.g., the converse function) instead of the binary function *intersection*, see the `hom1-1` method in section 3.2.5. To handle situations involving a binary function as discussed above, the `hom1-1` method has to be adapted to `hom1-2`. One potential criticism is that we should instead construct more general methods which cover large classes of problems. Although general methods are definitely needed for effective proof planning systems, this by no means excludes the need of modification. It is very difficult, for instance, to come up with a single method covering all possible cases which a human mathematician would *intuitively* consider as an example of homomorphy.

While in Bundy's approach the specification is declarative, the tactic itself is still procedural. Hence a method cannot be automatically adapted to new situations. However, the strength of human reasoning and problem solving depends to a great extent on the ability to adapt existing problem solving facilities to related, but not directly fitting situations. In order to allow such an automated modification of methods, we have proposed in [18] a separation of the tactic into a declaratively and a procedurally represented part. Though the separation of procedural and declarative knowledge is widespread in AI systems in general, it is not the case in most of the existing automated reasoning systems. As shown in [18] this separation leads not only to more natural methods, but practically enables the formulation of general meta-level mechanisms which adapt existing methods to suit novel situations. In this way, an automated modification needs only to be performed on the declarative part, hence it is desirable to store most of the relevant information in this part. Only the rest should be encoded in the procedural part. Preferably the procedural part is always one of few standard interpreters evaluating the declarative part. However, although methods consisting of only a procedural part cannot be adapted, they are not excluded in our framework. The work of Giunchiglia and Traverso [14] to represent tactics in a logical meta-language has a similar motivation as our work, namely to represent tactics in a declarative manner in order to allow a mechanical modification. In their approach the whole tactic is represented on a logical meta-level, what enables a full declarative representation. In our approach only parts of the tactic are represented declaratively, what should enable easier transformations in some cases.

In the next section we summarize the fundamentals for our approach, that is, the logic, the natural deduction calculus, and the notion of a tactic we want to employ in the

following. In chapter 3 we introduce the key concept of a method. In chapter 4 we show how the methods can be used in a planning framework. In the following chapter we discuss how methods can be adapted by so-called meta-methods. After the conclusion we present an annotated run-time protocol of an example in the appendix.

Chapter 2

Logical Foundations: Calculus and Tactics

Mein teurer Freund, ich rat' Euch drum
Zuerst Collegium Logicum

Johann Wolfgang Goethe, Faust I

In this chapter we introduce the basic machinery, in particular the underlying logic $\mathcal{POST}$, our proof format, and the definition of the notion tactic.

2.1 The Higher-Order Language $\mathcal{POST}$

A fundamental question to be answered when representing and proving mathematical theorems concerns the underlying logic. Since the technical mathematical language of a typical textbook is essentially a sorted higher-order logic augmented by many special-purpose representational constructs that are typical for the field at hand, for our proof development environment Ω-MKRP [17] a corresponding language called $\mathcal{POST}$ [19, 20] has been developed. Since we are mainly interested in applications of standard mathematics, we adopt the *classical* higher-order logic as opposed to a non-standard logic such as intuitionistic logic. In particular, our logic is built on Church's simple theory of types [8] (for an excellent introduction to classical higher-order logic, see [2]), enriched by sorts (similar to the way first-order logic is extended to sorted first-order logic).

To keep the presentation simple, we restrict ourselves in this report to an unsorted higher-order logic as object language, although we think a sorted version to be much more adequate for the practice of mathematical reasoning.

Formally, we consider $\mathcal{POST}$ to be a standard higher-order language as described in [2]. Each term of $\mathcal{POST}$ has a type, where types are the simple types as introduced by Church [8]. There are the basic types ι and o standing for the individuals and the truth values on the one hand. On the other hand functional types are built in the following way: if $\tau_1, \ldots, \tau_n, \sigma$ are types then $(\tau_1 \times \cdots \times \tau_n \rightarrow \sigma)$ is the type of the functions from $\tau_1 \times \cdots \times \tau_n$ to σ.

Each constant symbol or variable symbol of a certain type is a term of this type. General terms are built up from these by applications and abstractions. If $t_1, \ldots, t_n$ are terms of

types $\tau_1, \ldots, \tau_n$ and f is a function term of type $(\tau_1 \times \cdots \times \tau_n \rightarrow \sigma)$, then $f(t_1, \ldots, t_n)$, the *application* of f to its arguments, is a term of type σ.

If t is a term of type σ and $x_1, \ldots, x_n$ are variables of type $\tau_1, \ldots, \tau_n$, then the expression $\lambda x_1, \ldots, x_n . t$, the *abstraction* of t by $x_1, \ldots, x_n$, is a term of type $(\tau_1 \times \cdots \times \tau_n \rightarrow \sigma)$. In the usual way the quantifiers can be defined using the λ-binder, that is, $\forall x.\varphi$ is an abbreviation for $\Pi\lambda x.\varphi$, with a polymorphic[1] constant Π that stands for a function with (truth) value t if the argument function is true for all its possible input. $\exists x.\varphi$ is an abbreviation for $\neg\forall x.\neg\varphi$. Instead of the type theoretic notion $P(x)$ or $Q(x, y)$ we often use the set-theoretic notion $x \in P$ or $\langle x, y\rangle \in Q$ as a syntactic variant, but nevertheless always stay within type theory.

Each term of type o is called a formula. We assume special function constants in our logic standing for the connectives, namely $\neg$ of type $(o \rightarrow o)$ and $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$ of type $(o \times o \rightarrow o)$. In order to indicate the type of a term, we sometimes write it as an index of the term, for instance, x_ι, $\neg_{(o\rightarrow o)}$, $f_{(\iota\rightarrow\iota)}$.

2.2 The Natural Deduction Proof Formalism

The natural deduction (ND) calculus first proposed by Gentzen in [12, 13] is adopted as the basic calculus. Gentzen called his system *natural* deduction, since the "*inference rules of the system of natural deduction correspond closely to procedures common in intuitive reasoning, and when informal proofs ... are formalized within these systems, the main structure of the informal proofs can often be preserved.*" [25]. Concretely, we adopt a linearized version of ND proofs introduced in [1]. In this formalism, an ND proof is a sequence of *proof lines*, each of them is of the form:

$$\textit{Label} \quad \triangle \quad \vdash \quad \textit{Derived-Formula} \qquad (\textit{Rule} \quad \textit{premise-lines})$$

where *Rule* is restricted to a rule of inference in ND, which justifies the derivation of the *Derived-Formula* using formulae in *premise-lines*. *Rule* and *premise-lines* together are called the justification of a line. $\triangle$ is a finite set of formulae, being hypotheses the derived formula depends on. Since a natural deduction proof can also be viewed as a proof tree, we will talk about proof trees as well.

The set of elementary inference rules we choose are basically those identified by Gentzen. We adopt his calculus $\mathcal{NK}$, plus a pair of symmetric rules handling the disjunction ($\vee E_l$ and $\vee E_r$). The collection of elementary rules adopted is listed in table 2.1.

Every figure in Table 2.1 represents an inference rule. Proof line schemata (proof lines containing meta-variables, with justifications omitted) separated by commas above the bar represent preconditions. Meta-variables F, G, and H can be substituted by any formula, $\forall x.F_x$, $\exists x.F_x$ by any formula with $\forall$ or $\exists$ as the top symbol, where "x" denotes the corresponding bound variable. F_a denotes the formula achieved by replacing all free occurrences of the variable "x" in F_x by an individual constant "a". The meta-variable "a" in $\forall E$ can be substituted by an arbitrary term. For rule $\forall I$ and *CHOICE*, in addition, the following *variable conditions* must be checked:

[1] Actually, we employ not only type constants, but also type variables with type inference. Since this does not play a major role in this paper, the reader need not bother about this subtlety of the object language.

Structural Gentzen Rules:

$$\frac{}{\triangle, F \vdash F}\,Hyp, \qquad \frac{\triangle, F \vdash G}{\triangle \vdash F \to G}\,Ded, \qquad \frac{\triangle \vdash \exists x.F_x,\ \triangle, F_a \vdash H,}{\triangle \vdash H}\,Choice,$$

$$\frac{\triangle \vdash F \vee G,\ \triangle, F \vdash H,\ \triangle, G \vdash H}{\triangle \vdash H}\,CASE, \qquad \frac{\triangle, G \vdash \bot}{\triangle \vdash \neg G}\,IP_1, \qquad \frac{\triangle, \neg G \vdash \bot}{\triangle \vdash G}\,IP_2$$

Non-Structural Gentzen Rules

$$\frac{\triangle \vdash F,\ \triangle \vdash G}{\triangle \vdash F \wedge G}\wedge I, \quad \frac{\triangle \vdash F}{\triangle \vdash F \vee G}\vee I_l, \quad \frac{\triangle \vdash G}{\triangle \vdash F \vee G}\vee I_r, \quad \frac{\triangle \vdash F_a}{\triangle \vdash \forall x.F_x}\forall I \quad \frac{\triangle \vdash F_a}{\triangle \vdash \exists x.F_x}\exists I$$

$$\frac{\triangle \vdash F \wedge G}{\triangle \vdash F}\wedge E_l, \qquad \frac{\triangle \vdash F \wedge G}{\triangle \vdash G}\wedge E_r, \qquad \frac{\triangle \vdash F,\ \triangle \vdash F \to G}{\triangle \vdash G} \to E,$$

$$\frac{\triangle \vdash P \vee Q,\ \triangle \vdash \neg Q}{\triangle \vdash P}\vee E_r, \qquad \frac{\triangle \vdash P \vee Q,\ \triangle \vdash \neg P}{\triangle \vdash Q}\vee E_l, \qquad \frac{\triangle \vdash \forall x.F_x}{\triangle \vdash F_a}\forall E$$

$$\frac{\triangle \vdash F,\ \triangle \vdash \neg F}{\triangle \vdash \bot}\neg E, \qquad \frac{\triangle \vdash \bot}{\triangle \vdash D}\bot \qquad \frac{\triangle \vdash \neg(\neg F)}{\triangle \vdash F}\neg\neg$$

Table 2.1: Elementary Inference Rules

- The variable condition for $\forall I$: meta-variable "a" (Eigenvariable in [13]) may not occur in $\forall x.F_x$ or in any formula in the assumption set $\triangle$.
- The variable condition for *CHOICE*: meta-variable "a" may not occur in H, or in any formula in the assumption set $\triangle$.

In order to uniformly represent proof plans, we extend the ND proof formalism by allowing the *Rule* slot to be replaced by the name of a method or simply the value "OPEN". If a line is justified by a method, it is a part of the current proof plan. Open lines are still to be justified in the planning process. $A \leftrightarrow B$ is a shorthand for $A \to B \wedge B \to A$.

Below is an example, containing two OPEN lines as pending goals awaiting to be fulfilled.

1. 1	$\vdash$	$\forall\sigma.\forall x, y.\langle x, y\rangle \in \text{converse}(\sigma) \leftrightarrow \langle y, x\rangle \in \sigma$	(Hyp)
2. 2	$\vdash$	$\forall\sigma.\text{symmetric}(\sigma) \leftrightarrow \forall x, y.\langle x, y\rangle \in \sigma \to \langle y, x\rangle \in \sigma$	(Hyp)
3. 3	$\vdash$	$\text{symmetric}(\rho)$	(Hyp)
4. 2	$\vdash$	$\text{symmetric}(\rho) \leftrightarrow \forall x, y.\langle x, y\rangle \in \rho \to \langle y, x\rangle \in \sigma$	($\forall E$ 2)
5. 2	$\vdash$	$\text{symmetric}(\rho) \to \forall x, y.\langle x, y\rangle \in \rho \to \langle y, x\rangle \in \sigma$	($\wedge E_l$ 4)
6. 2,3	$\vdash$	$\forall x, y.\langle x, y\rangle \in \rho \to \langle y, x\rangle \in \rho$	($\to E$ 3 5)
7. 1,2,3	$\vdash$	$\forall x, y.\langle x, y\rangle \in \text{converse}(\rho) \to \langle y, x\rangle \in \text{converse}(\rho)$	(OPEN 1 6)
8. 1,2,3	$\vdash$	$\text{symmetric}(\text{converse}(\rho))$	(OPEN 2 7)

2.3 Tactics

As indicated in the introduction, it can be very cumbersome and tedious for a user to construct a proof at the calculus level. In order to overcome this, the concept of a *tactic* was proposed by Milner et al. [15]. Basically tactics can be seen as derived calculus rules. Technically, they are inductively defined as basic calculus level rules or as the application of so-called tacticals to existing tactics.

The view above is too restrictive to including more complicated tactics, therefore in our framework a *tactic* is a function that generates new proof lines and inserts them to the current proof. Following the declarative approach proposed in [18], this function is represented in two parts. One part is a set of proof line schemata, that is, proof lines with meta-variables. The other part contains a procedure. The whole tactic can then be seen as a function with parameters. An application of this function with concrete instances for the meta-variables generates new proof lines by applying the procedure to the proof line schemata.

Most commonly the procedure is just one standard interpreter, which basically instantiates proof line schemata by binding meta-variables. In other cases, the procedure can be a sophisticated theorem prover. Hence the range of possible tactics is very wide, reaching from the application of an ND rule to the call of an incorporated theorem prover. Since we allow arbitrary proof lines to be added to the current proof state, the correctness of the final proof is not ensured a priori, but must be checked by a verifier.

Chapter 3

A Declarative Approach toward Methods

> Studiorum finis esse debet ingenii directio ad solida et vera, de iis omnibus quae occurrunt, proferenda judicia.
>
> *René Descartes, Regulæ ad Directionem Ingenii*

3.1 The General Notion of Method

A central concept of knowledge based reasoning in mathematics 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.

In our framework a method can basically be divided into a declarative and a procedural part. By discerning the declarative part of a method, it is now possible to formulate meta-level methods adapting the declarative part of existing methods and thus come up with novel ones.

Concretely, we define a method as consisting of the following slots:

- Declarations: A signature that declares meta-variables used in the method,
- Premises: Schemata of proof lines which are used by this method as assumptions,
- Conclusions: Schemata of proof lines which this method is designed to prove,
- Constraint: A formula in the constraint language to be described in section 3.2.3. This is used to formulate further restrictions on the premises and the conclusions, which can not be formulated in terms of proof line schemata.
- Tactic: It is split into two components:

- Declarative content: This slot is currently restricted to schemata of partial proofs. For pure procedural methods, it can be empty.
- Procedural content: This slot contains a procedure which produces subproofs connecting the premises and the conclusions of the method using the declarative content.

The procedural part of a method ranges from a simple standard interpreter which basically instantiates the declarative part of the tactic to a complex reasoning procedure working by itself.

Some methods only suggest a proof sketch. They produce possible subgoals without fully specifying the way for arriving at the conclusions from the premises or vice versa.

3.2 Formal Definitions

In the next section we first formally define the syntax of a method. The corresponding semantics is explained then in section 3.2.2.

3.2.1 Syntax of a Method

In this section we will concentrate on the description of the components which can be easily explained: the declarations, the premises, the conclusions, and the tactic with its declarative and its procedural part. For the sake of completeness we will only mention the last, more complex component, the constraint, and devote section 3.2.3 to it. The section is concluded by a formal definition of the notion of a method.

3.2.1.1 Declarations

A method normally contains meta-variables that are instantiated upon application. In order to distinguish constants and meta-variables, all meta-variables have to be declared in the declaration slot. A declaration for a meta-variable is a variable symbol followed by a sort symbol. Formally it is a set

$$D = \{x{:}\mathsf{s} \mid x \in \mathbf{V}^{\mathcal{M}} \text{ and } \mathsf{s} \in \mathbf{SORT}^{\mathcal{M}}\}$$

where $\mathbf{V}^{\mathcal{M}}$ is the set of meta-variables and $\mathbf{SORT}^{\mathcal{M}}$ is the set of sort symbols as defined in definition 3.2 below.

3.2.1.2 Tactic

In our model a *tactic* is split into two parts: An ND proof schema with meta-variables, called the *declarative content*, and a piece of program, called the *procedural content*. This procedural part can be a standard interpreter which creates new proof lines by instantiating the declarative content and then inserts them into the current proof state. It can also be an arbitrary piece of procedural knowledge, e.g., an automated theorem prover.

Formally a tactic is a pair $\langle T^{decl}, T^{proc}\rangle$, where T^{decl} is a finite list of schematic proof lines of the form $(\alpha_i) \quad H_i \vdash F_i \quad J_i$, or a meta-variable standing for such a list. Here the α_i are labels of proof lines. The H_i are either a list of proof line labels standing for the formulae

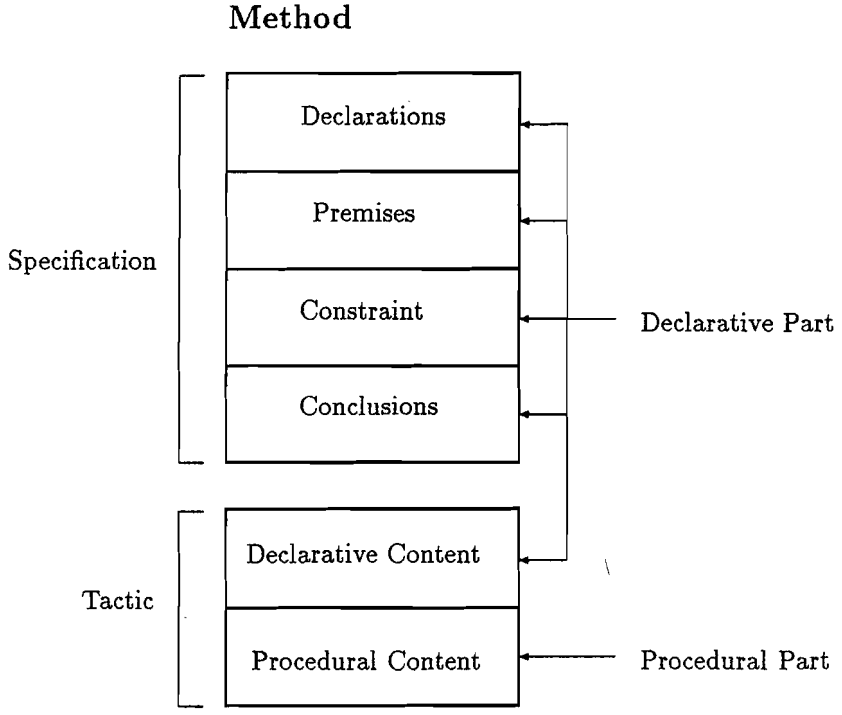


Figure 3.1: The Structure of Methods

of these lines or a meta-variable representing such a list. The F_i are formula schemata, i.e., formulae containing meta-variables. The J_i are justifications or the corresponding meta-variables. T^{proc} is a program, which generates new proof lines by interpreting T^{decl}.

3.2.1.3 Premises and Conclusions

Intuitively, the *premises* slot contains a list of proof line schemata which are used to prove the lines in the *conclusions* slot. In most cases both slots are subsets of the proof lines in the declarative content of the tactic. The proof lines can be marked with an additional sign, either "$\oplus$" or "$\ominus$". These signs have no effect on the constraint language and play only a role in the planning process (compare chapter 4).

3.2.1.4 Constraint

The more complex applicability condition of a method is formulated in the *constraint*. We devote section 3.2.3 to the constraint language.

3.2.1.5 The Definition of a Method

Now we can formally define the notion of a *method* which is informally introduced above.

Definition 3.1: A *method* M is a 6-tuple

$$M = \left\langle D, Pre, C, Con, T^{decl}, T^{proc} \right\rangle,$$

The components of the method are:

- D is the set of variable declarations,
- Pre and Con are the premises and conclusions,
- C is a formula of the constraint language $\mathcal{CL}$ representing the constraint of the method M,
- T^{decl} and T^{proc} is the declarative content and the procedural content of the tactic of the method M.

3.2.2 Semantics of a Method

A method consists of two major parts: the specification and the tactic (cf. figure 3.1). Since the specification is used in planning and the tactic in the execution of the plan operator, a method's semantics can also be divided into two parts: its semantics as a plan operator and as a tactic. The semantics as a plan operator is defined in chapter 4, where STRIPS-like plan operators are constructed from the specification.

The semantics as a tactic was already described in section 2.3, with the meta-variable bindings resulting from the planning process the procedure can be applied to the instantiated declarative content constructing new proof lines.

3.2.3 The Constraint Language $\mathcal{CL}$

As described previously, the applicability of a method is in the first place specified in terms of schemata for the premises and the conclusions. Furthermore, the constraint slot contains additional meta-level application conditions.

First we motivate our definition of the constraint language by discussing some criteria for an effective specification language in a proof planning environment:

1. *Expressiveness:* Since we want to check the applicability of methods, in our language it must be possible to express all properties of the objects in the current proof state.
2. *Adaptability:* The specification language should support automatic modification of methods by meta-methods.
3. *Tractability:* Since the specifications play the role of plan operators in a proof planning environment and the applicability of an operator must be effectively computable, the specification language must be decidable.
4. *Structured Representation:* The specification language should not only allow to formulate decidable application conditions of methods, but it should also be efficiently computable. Therefore the conditions should be structured in order to check the most important conditions first.

There are several approaches to relate the object language and the meta-language: Both languages can overlap, the object language can be included into the meta-language (see e.g. [26]), or both can be strictly separated from each other. Here we follow the third approach. While the object logic is a variant of the simple type theory as described in section 2.1, our constraint language is a variant of first-order predicate logic. Our language fulfills the first criterion, since it includes a broad range of functions and predicates manipulating the entities of the underlying object level.

The second criterion for the specification language is achieved since the constraint is kept declarative. Furthermore we have a clearly defined model-theoretic semantics to describe the applicability of methods, so that meta-methods can operate on the constraint taking the applicability criteria into consideration.

In order to guarantee the third criterion, the decidability, the following restrictions are imposed on our constraint language: First we only allow quantification over *finite lists* of objects of the object language, i.e., terms, types, proof lines, inference rules, justifications, positions, and substitutions. Second all function and predicate symbols are assigned a fixed interpretation that can be effectively computed. To further improve efficiency the different categories of objects of the object level are reflected as different sorts at the meta-level to avoid unnecessary instantiations.

The fourth criterion is not achieved in the constraint language itself. Rather it is achieved by separating the schemata of proof lines and the logical restrictions formulated in the constraint language, so we have a structured formalism for representing plan operators.

We also have two other concepts deviating from the standard definition of logics: Firstly we need a binding mechanism. As we have mentioned, the free meta-variables of the method are bound by the planner via matching. But it is also possible to assign a value to remaining unbound ones. This can be necessary, for instance when a new formula should be constructed by evaluating the constraint. Therefore we include a binding mechanism "$\leftarrow$", which is interpreted as a combination of a predicate and an assignment known from procedural languages. We describe it in definition 3.6. The second non-standard constituent is the construct **eval** specifying the applicability of a method when only an execution of the tactic binds all meta-variables.

3.2.3.1 The Syntax of $\mathcal{CL}$

Based on the discussion above, the syntax of our constraint language is a sorted first-order language with fixed function declarations. Since the primary use of the language is to make meta-level statements over proof objects used in a method at the object level, we have non-standard constituents in our syntax: The different categories of objects at the object level are reflected as different sorts at the meta-level, for instance, terms of the object level are reflected as meta-level constants of sort **term**. We will only allow quantifications over finite lists because the method language should be decidable.

In the following the general concepts of a sorted first-order language are described. The concrete restrictions to guarantee the tractability are introduced later.

We denote a symbol representing a meta-object with an $\mathcal{M}$ in its superscript.

Definition 3.2: A sorted *signature* $\Sigma^{\mathcal{M}}$ for the meta-level of a language is an 8-tuple

$$\langle \mathbf{V}^{\mathcal{M}}, \mathbf{F}^{\mathcal{M}}, \mathbf{P}^{\mathcal{M}}, \mathbf{SORT}^{\mathcal{M}}, \mathfrak{s}, \mathbf{SD}^{\mathcal{M}}, \mathbf{FD}^{\mathcal{M}}, \mathbf{PD}^{\mathcal{M}} \rangle, \qquad \text{where}$$

- $\mathbf{V}^{\mathcal{M}}$ is a set of variables,
- $\mathbf{F}^{\mathcal{M}}$ is a set of function symbols,
- $\mathbf{P}^{\mathcal{M}}$ is a set of predicate symbols,
- $\mathbf{SORT}^{\mathcal{M}}$ is a set of sort symbols,
- $\mathfrak{s}$ is a function $\mathfrak{s}: \mathbf{V}^{\mathcal{M}} \rightarrow \mathbf{SORT}^{\mathcal{M}}$, which assigns a variable symbol $v \in \mathbf{V}^{\mathcal{M}}$ a sort symbol $\mathfrak{s}(v) \in \mathbf{SORT}^{\mathcal{M}}$. Rather than $\mathfrak{s}(v) = \mathsf{s}$, we use the notation $v{:}\mathsf{s}$,
- $\mathbf{SD}^{\mathcal{M}}$ is the set of subsort declarations of the form $\mathsf{s}_1 \sqsubseteq \mathsf{s}_2$ where $\mathsf{s}_1, \mathsf{s}_2 \in \mathbf{SORT}^{\mathcal{M}}$,
- $\mathbf{FD}^{\mathcal{M}}$ is a set of function declarations.
  A *function declaration* is a pair $\langle f, \mathsf{s}_1 \times \cdots \times \mathsf{s}_n \rightarrow \mathsf{s} \rangle$ written as $f{\Leftarrow}\mathsf{s}_1 \times \cdots \times \mathsf{s}_n \rightarrow \mathsf{s}$ where $f \in \mathbf{F}^{\mathcal{M}}$ and $\mathsf{s}, \mathsf{s}_i \in \mathbf{SORT}^{\mathcal{M}}$ for $1 \leq i \leq n$.
- $\mathbf{PD}^{\mathcal{M}}$ is a set of predicate declarations.
  A *predicate declaration* is a pair $\langle P, \mathsf{s}_1 \times \cdots \times \mathsf{s}_m \rangle$ written as $P{\Leftarrow}\mathsf{s}_1 \times \cdots \times \mathsf{s}_m$ with $P \in \mathbf{P}^{\mathcal{M}}$ and $\mathsf{s}_i \in \mathbf{SORT}^{\mathcal{M}}$ for $1 \leq i \leq m$. Different from other predicates, the equality $\doteq$ is defined for every sort, i.e. $\doteq{\Leftarrow}\mathsf{s} \times \mathsf{s}$. Another special construct is the binding predicate $\leftarrow$, which is defined for every sort.

We assume infinitely many variables in each sort.

Example: $\Sigma^{\mathcal{M}}$ is a sorted signature, if

- the set of variables $\mathbf{V}^{\mathcal{M}} = \{v_i \mid i \in \mathbb{N}\}$,
- the set of constants $\mathbf{F}^{\mathcal{M}} = \{g, f, c\}$ with the constants g of arity 2, f of arity 1 and c of arity 0,
- the set of predicates $\mathbf{P}^{\mathcal{M}} = \{\doteq\}$,
- the set of sorts $\mathbf{SORT}^{\mathcal{M}} = \{\mathsf{var}, \mathsf{const}, \mathsf{term}\}$,
- the variables with the sorts $\mathfrak{s}(v_1) = \mathsf{var}$, $\mathfrak{s}(v_2) = \mathsf{const}$, $\mathfrak{s}(v_3) = \mathsf{term}$, etc.,
- the subsort declarations $\mathbf{SD}^{\mathcal{M}} = \{\mathsf{var} \sqsubseteq \mathsf{term}, \mathsf{const} \sqsubseteq \mathsf{term}\}$ declaring that constants and variables are subsorts of terms,
- the function declarations $\mathbf{FD}^{\mathcal{M}} = \{g{\Leftarrow}\mathsf{term} \times \mathsf{term} \rightarrow \mathsf{term}, f{\Leftarrow}\mathsf{term} \rightarrow \mathsf{term}, c{\Leftarrow}\mathsf{const}\}$ and
- the predicate declarations $\mathbf{SD}^{\mathcal{M}} = \{\doteq{\Leftarrow}\mathsf{var} \times \mathsf{var}, \doteq{\Leftarrow}\mathsf{const} \times \mathsf{const}, \doteq{\Leftarrow}\mathsf{term} \times \mathsf{term}\}$.

Now we define the sorted terms of our language as usual.

Definition 3.3: Let $\Sigma^{\mathcal{M}}$ be a sorted signature. The *terms of sort* s for the signature $\Sigma^{\mathcal{M}}$, designated by $\mathbf{T}^{\mathcal{M}}_{\mathsf{s}}$, are inductively defined as follows:

(i) $v \in \mathbf{T}^{\mathcal{M}}_{\mathsf{s}}$, if $\mathfrak{s}(v) \sqsubseteq \mathsf{s}$,

(ii) $f(t_1,\ldots,t_n) \in \mathbf{T}_{\mathsf{s}}^{\mathcal{M}}$, if f has a function declaration $f \Leftarrow \mathsf{s}_1 \times \cdots \times \mathsf{s}_n \rightarrow \mathsf{r}$, $\mathsf{r} \sqsubseteq \mathsf{s}$ and $t_i \in \mathbf{T}_{\mathsf{s}_i}^{\mathcal{M}}$ for $1 \leq i \leq n$.

The set of all *terms* is defined as $\mathbf{T}^{\mathcal{M}} = \bigcup_{\mathsf{s} \in \mathbf{SORT}^{\mathcal{M}}} \mathbf{T}_{\mathsf{s}}^{\mathcal{M}}$.

Example: Suppose the sorted signature is defined as in the example above. Then we have $v_3 \in \mathbf{T}_{\mathsf{term}}^{\mathcal{M}}$, therefore $f(v_3) \in \mathbf{T}_{\mathsf{term}}^{\mathcal{M}}$ because of the function declaration for f, $f \Leftarrow \mathsf{term} \rightarrow \mathsf{term}$. Also $c \in \mathbf{T}_{\mathsf{term}}^{\mathcal{M}}$, because of $\mathsf{const} \sqsubseteq \mathsf{term}$, and finally, when we apply the function declaration of g, $g \Leftarrow \mathsf{term} \times \mathsf{term} \rightarrow \mathsf{term}$ to $f(v_3)$ and c, we can conclude that $g(f(v_3), c)$ is in $\mathbf{T}_{\mathsf{term}}^{\mathcal{M}}$.

Now we define the set of well-formed formulae. Our definition of a formula differs from the standard one, since we allow only quantification over finite lists of terms and because of the special binding predicate "$\leftarrow$".

Definition 3.4: The set of *well-formed formulae*, $\mathbf{WFF}^{\mathcal{M}}$, for a given signature $\Sigma^{\mathcal{M}}$ is inductively defined (In order to distinguish connectives and quantifiers in the constraint language from those in the object language different symbols are used). The logic connectives and quantifiers are listed in groups (iii)–(v) in decreasing order of the binding priority.

(i) If $P \in \mathbf{P}^{\mathcal{M}}$, $P \Leftarrow \mathsf{s}_1 \times \cdots \times \mathsf{s}_n$ is a sort declaration for the predicate symbol $P \in \Sigma^{\mathcal{M}}$, and $t_i \in \mathbf{T}_{\mathsf{s}_i}^{\mathcal{M}}$ for $1 \leq i \leq n$, then $P(t_1,\ldots,t_n) \in \mathbf{WFF}^{\mathcal{M}}$. Also $\top, \bot \in \mathbf{WFF}^{\mathcal{M}}$, $\top$ stands for truth and $\bot$ for falsehood.

(ii) If $x \in \mathbf{V}^{\mathcal{M}}$ with the sort s and $t \in \mathbf{T}_{\mathsf{s}}^{\mathcal{M}}$, then $(x \leftarrow t) \in \mathbf{WFF}^{\mathcal{M}}$.

(iii) If $\Phi \in \mathbf{WFF}^{\mathcal{M}}$, then $(\sim \Phi) \in \mathbf{WFF}^{\mathcal{M}}$.

(iv) If $\Phi_1, \Phi_2 \in \mathbf{WFF}^{\mathcal{M}}$, then $(\Phi_1 \circ \Phi_2) \in \mathbf{WFF}^{\mathcal{M}}$, for $\circ \in \{\&, |\}$.

(v) If $t \in \mathbf{T}_{\mathsf{list(s)}}^{\mathcal{M}}$, $x{:}\mathsf{s} \in \mathbf{V}^{\mathcal{M}}$ and $\Phi \in \mathbf{WFF}^{\mathcal{M}}$, then $(\bigwedge^t x{:}\mathsf{s}.\Phi) \in \mathbf{WFF}^{\mathcal{M}}$ and $(\bigvee^t x{:}\mathsf{s}.\Phi) \in \mathbf{WFF}^{\mathcal{M}}$.

The implication $\Phi_1 \Rightarrow \Phi_2$ is a short-hand for $\sim \Phi_1 \mid \Phi_2$.

Example: With the terms given above and $s(v_4) = \mathsf{list(term)}$, we can build the formula:

$$\bigwedge^{v_4} z{:}\mathsf{term}.c \doteq z \mid \sim c \doteq g(f(z), c).$$

3.2.3.2 The Semantics of $\mathcal{CL}$

The semantics of $\mathcal{CL}$ describes the applicability criterion of a method. Before giving formal definitions, let us first examine the semantics of the binary binding predicate "$\leftarrow$" which is basically a non-logical construct. The semantics of "$\leftarrow$" should consider the following: First we do not want to alter the binding resulting from the matching in the planning process,

since we assume that this has a higher priority than those specified in the constraint. In other words only variables not bound by this matching should be bound by the constraint. Second a disjunction should work like a case analysis with respect to the bindings, that is, all bindings made by "$\leftarrow$" in a left side of a disjunction should be removed form the actual binding when interpreting the right-hand side. In contrast, a conjunction should propagate the bindings to all subparts and the remaining constraint.

Furthermore, semantics is only defined for closed formulae. Technically speaking this means that all variables in a constraint must be bound either by a quantification or by an assignment. A method is not applicable when an unbound variable is encountered during applicability check.

First we give the definition of an *interpretation*.

Definition 3.5: Let $\Sigma^{\mathcal{M}}$ be a signature. A $\Sigma^{\mathcal{M}}$-*interpretation* $\mathbf{I}$, is a pair $\langle \mathcal{D}, \mathcal{I} \rangle$ where $\mathcal{D}$ is a nonempty set (to be defined later) called the *domain* of $\mathbf{I}$, and $\mathcal{I}$ is an *interpretation function* that assigns:

- to every function symbol $f \in \mathbf{F}^{\mathcal{M}}$ of arity n a function $f^{\mathcal{I}} \in \mathcal{D}^n \rightarrow \mathcal{D}$,
- to every predicate symbol $p \in \mathbf{P}^{\mathcal{M}}$ of arity n a predicate $p^{\mathcal{I}} \in \mathcal{D}^n$.

For every sort $\mathsf{s} \in \mathbf{SORT}^{\mathcal{M}}$ exists a non-empty set $\mathcal{D}_{\mathsf{s}}$. The domain $\mathcal{D}$ is the union of all these domains $\mathcal{D}_{\mathsf{s}}$, i.e. $\mathcal{D} = \bigcup_{\mathsf{s} \in \mathbf{SORT}^{\mathcal{M}}} \mathcal{D}_{\mathsf{s}}$. Furthermore $\mathcal{D}_{\mathsf{r}} \subseteq \mathcal{D}_{\mathsf{s}}$, if $\mathsf{r} \sqsubseteq \mathsf{s}$ is a subsort declaration. We also assume that if a term t has sort s, then $t^{\mathcal{I},\varphi}$ is in $\mathcal{D}_{\mathsf{s}}$. A well-sorted term $f(t_1, \ldots, t_n)$ should satisfy that $(t_1^{\mathcal{I},\varphi}, \ldots, t_n^{\mathcal{I},\varphi})$ is in the domain of $f^{\mathcal{I}}$.

Deviating from the standard definition of a value of a formula Ψ under an interpretation function $\mathcal{I}$ and an assignment φ, denoted by $\Psi^{\mathcal{I},\varphi}$, which is usually a truth value in $\{\mathsf{t}, \mathsf{f}\}$, we define it as a truth-value assignment pair in $\{\mathsf{t}, \mathsf{f}\} \times \mathcal{F}_{\mathsf{p}}(\mathbf{V}^{\mathcal{M}}, \mathcal{D}_{\mathsf{s}})$, where $\mathcal{F}_{\mathsf{p}}(\mathbf{V}^{\mathcal{M}}, \mathcal{D}_{\mathsf{s}})$ is the set of all partial functions from $\mathbf{V}^{\mathcal{M}}$ to $\mathcal{D}_{\mathsf{s}}$. The second component of such pairs is needed to keep track of the bindings.

The following definition specifies the semantics of the constraint language. Recall that this language is used to formulate the applicability condition of a method. Starting from an assignment φ given by the planner, all bindings are accumulated in the second component of the truth-value assignment pairs. The method is *applicable* when the interpretation of the constraint results in t in the first component and no variable specified in the declaration slot remains unbound, otherwise it is not applicable. In the following we denote the domain of the partial function φ, where φ is defined, by $\mathrm{Dom}(\varphi)$, $\mathrm{Var}(t)$ denotes the set of all variables of a term t.

Definition 3.6: Let φ be an assignment, which is a partial function mapping variables of sort s to elements of $\mathcal{D}_{\mathsf{s}}$. We define the *value* of a formula Ψ under the assignment φ and the interpretation function $\mathcal{I}$, denoted by $\Psi^{\mathcal{I},\varphi}$, recursively.

The value of the terms are interpreted as usual in the standard Tarskian model-theoretic semantics:

(1) When Ψ is a term $f(t_1, \ldots, t_n)$:

$$\Psi^{\mathcal{I},\varphi} = f^{\mathcal{I}}(t_1^{\mathcal{I},\varphi}, \ldots, t_n^{\mathcal{I},\varphi}),$$

(2) When Ψ is an atom $p(t_1, \ldots, t_n)$ and p is not the assignment predicate "$\leftarrow$":

$$\Psi^{\mathcal{I},\varphi} = \begin{cases} \langle \mathsf{t}, \varphi \rangle, \text{ if } \left\langle t_1^{\mathcal{I},\varphi}, \ldots, t_n^{\mathcal{I},\varphi} \right\rangle \in p^{\mathcal{I}}, \\ \langle \mathsf{f}, \varphi \rangle, \text{ otherwise.} \end{cases}$$

(3) When Ψ is a variable x of sort s:

$$\Psi^{\mathcal{I},\varphi} = \varphi(x) \in \mathcal{D}_{\mathsf{s}},$$

(4) When Ψ has the form $(x \leftarrow t)$:

$$\Psi^{\mathcal{I},\varphi} = \begin{cases} [(x \doteq t)]^{\mathcal{I},\varphi} & \text{if } x \in \mathrm{Dom}(\varphi), \\ \left\langle \mathsf{t}, \varphi \cup \{x \mapsto t^{\mathcal{I},\varphi}\} \right\rangle & \text{if } x \notin \mathrm{Var}(t) \text{ and } x \notin \mathrm{Dom}(\varphi) \\ & \quad \text{and } \mathrm{Var}(t) \subseteq \mathrm{Dom}(\varphi), \\ \langle \mathsf{f}, \varphi \rangle & \text{otherwise.} \end{cases}$$

In the first case the assignment is interpreted as the equality predicate since the variable is already bound. A binding is given to a variable in the second case, which is the main goal of this construct. If the variable x recursively occurs in the term t, then the assignment cannot be properly performed.

(5) When Ψ has the form $\sim \Phi$:

$$\Psi^{\mathcal{I},\varphi} = \begin{cases} \langle \mathsf{f}, \varphi \rangle, \text{ if } \Phi^{\mathcal{I},\varphi} = \langle \mathsf{t}, \varphi \rangle, \\ \langle \mathsf{t}, \varphi \rangle, \text{ if } \Phi^{\mathcal{I},\varphi} = \langle \mathsf{f}, \varphi \rangle. \end{cases}$$

The value of the logical connectives & and | are determined by lazy evaluation:

(6) When Ψ has the form $\Phi_1 \,\&\, \Phi_2$:

$$\Psi^{\mathcal{I},\varphi} = \begin{cases} \Phi_2^{\mathcal{I},\varphi'}, \text{ if } \Phi_1^{\mathcal{I},\varphi} = \langle \mathsf{t}, \varphi' \rangle, \\ \langle \mathsf{f}, \varphi' \rangle, \text{ if } \Phi_1^{\mathcal{I},\varphi} = \langle \mathsf{f}, \varphi' \rangle. \end{cases}$$

(7) When Ψ has the form $\Phi_1 \mid \Phi_2$:

$$\Psi^{\mathcal{I},\varphi} = \begin{cases} \Phi_2^{\mathcal{I},\varphi}, \text{ if } \Phi_1^{\mathcal{I},\varphi} = \langle \mathsf{f}, \varphi' \rangle, \\ \langle \mathsf{t}, \varphi' \rangle, \text{ if } \Phi_1^{\mathcal{I},\varphi} = \langle \mathsf{t}, \varphi' \rangle. \end{cases}$$

Note that when the second part of a disjunction is interpreted, the bindings are restored to the bindings used before the first part was interpreted in order to undo the bindings made in the first part.

The quantifiers range over terms representing finite lists.

(8) When Ψ has the form $\bigvee^t x{:}\mathbf{s}.\Phi$:

$$\Psi^{\mathcal{I},\varphi} = \begin{cases} \langle \mathsf{t}, \varphi \rangle, \text{ if } t^{\mathcal{I},\varphi} = \langle d_1, \ldots, d_n \rangle, \\ \qquad \mathrm{Var}(t) \subseteq \mathrm{Dom}(\varphi), \\ \qquad x \notin \mathrm{Var}(t), \\ \qquad \{x \mapsto d_i\} \not\subseteq \varphi, 1 \leq i \leq n, \text{ and} \\ \qquad \text{there is a least index } i \text{ with } \Phi^{\mathcal{I},\varphi \cup \{x \mapsto d_i\}} = \langle \mathsf{t}, \varphi' \rangle \text{ for some } \varphi'. \\ \langle \mathsf{f}, \varphi \rangle \text{ otherwise.} \end{cases}$$

(9) When Ψ has the form $\bigwedge^t x{:}\mathbf{s}.\Phi$:

$$\Psi^{\mathcal{I},\varphi} = \begin{cases} \langle \mathsf{t}, \varphi \rangle, \text{ if } t^{\mathcal{I},\varphi} = \langle d_1, \ldots, d_n \rangle, \\ \qquad \mathrm{Var}(t) \subseteq \mathrm{Dom}(\varphi), \\ \qquad x \notin \mathrm{Var}(t), \\ \qquad \{x \mapsto d_i\} \not\subseteq \varphi, 1 \leq i \leq n, \text{ and} \\ \qquad \text{for all } i \text{ there is a } \varphi' \text{ such that } \Phi^{\mathcal{I},\varphi \cup \{x \mapsto d_i\}} = \langle \mathsf{t}, \varphi' \rangle. \\ \langle \mathsf{f}, \varphi \rangle \text{ otherwise.} \end{cases}$$

Example: For constructing a resolution proof by an automatic theorem prover the clause normal form of formulae must be computed. In this process it is necessary to push a negation, $\neg(\Phi_{21} \vee \Phi_{22})$, $\neg(\Phi_{21} \wedge \Phi_{22})$ into non-atomic formulae resulting in $\neg\Phi_{21} \wedge \neg\Phi_{22}$, $\neg\Phi_{21} \vee \neg\Phi_{22}$, respectively. This can be achieved using the **pushneg** method given in figure 3.2. Assume that in the planning process an assignment $\varphi = \{\Phi_1 \mapsto a \vee b\}$ is made, where a and b are two constants (The bindings for the other meta-variables H, J, etc. are omitted for simplicity). When interpreting the constraint of the **pushneg** method, the first three parts of the conjuction, that is, the test whether the formula is not atomic and the first two assignments result in $\langle \mathsf{t}, \varphi_1 \rangle$, where $\varphi_1 = \varphi \cup \{\Phi_{21} \mapsto \neg a, \Phi_{22} \mapsto \neg b\}$. Interpreting the first part of the disjunction results in $[h \leftarrow \vee \,\&\, \mathsf{conj}(\Phi_1)]^{\mathcal{I},\varphi_1} = \langle \mathsf{f}, \varphi_1 \cup \{h \mapsto \vee\} \rangle$, which means that the second part of the disjunction must be computed. It is evaluated with the binding φ_1, that is, the binding made in the first disjunction part, $\{h \mapsto \vee\}$, is not further considered. The interpretation of $[h \leftarrow \wedge \,\&\, \mathsf{disj}(\Phi_1)]^{\mathcal{I},\varphi_1}$ results in $\langle \mathsf{t}, \varphi_2 \rangle$, where $\varphi_2 = \varphi_1 \cup \{h \mapsto \wedge\}$, which means that the last part of the constraint must be interpreted: $[\Phi_2 \leftarrow \mathsf{application}(h, \langle \Phi_{21}, \Phi_{22} \rangle)]^{\mathcal{I},\varphi_2} = \langle \mathsf{t}, \varphi_2 \cup \{\Phi_2 \mapsto \neg a \wedge \neg b\} \rangle$. So the method is applicable and a new proof line with the formula $\neg a \wedge \neg b$ is inserted into the current proof. Note that the binding for the variable h was used while interpreting the last part of the constraint.

Now we examine the second non-standard construct **eval**. This is necessary since we allow more procedural methods, for which the effect cannot be adequately described in a declarative way. For instance, consider a procedure stripping off all universal quantifiers successively from the front of a given formula until the head of the formula is no more a universal quantifier. Here we have the constraint

$$\mathsf{all}(\mathsf{prInformula}(L_{pre})) \,\&\, \sim \mathsf{all}(\mathsf{prInformula}(L_{con})),$$

where L_{pre} is a meta-variable for the proof line in the premises of the method and L_{con} is a proof line in the conclusions. The meta-variable L_{con} cannot be bound until the procedural part of the tactic is executed.

Method : pushneg[1]	
Declarations	L_1, L_2:prln J:just h:const H:list(prln) Φ_{21}, Φ_{22}:term Φ_1, Φ_2:appl
Premises	$\ominus L_1$
Constraint	$\sim$ atom(Φ_1) & $\Phi_{21} \leftarrow \neg$1st(applargs($\Phi_1$)) & $\Phi_{22} \leftarrow \neg$2nd(applargs($\Phi_1$)) & $((h \leftarrow \vee$ & conj(Φ_1)) | $(h \leftarrow \wedge$ & disj(Φ_1))) & $\Phi_2 \leftarrow$ application($h, \langle \Phi_{21}, \Phi_{22} \rangle$)
Conclusions	$\oplus L_2$
Declarative Content	(L_1) $H \vdash \neg\Phi_1$ (J) (L_2) $H \vdash \Phi_2$ (pushneg L_1)
Procedural Content	schema – interpreter

Figure 3.2: The pushneg method

There are similar cases, for instance when the underlying procedural tactic carries out indeterministic operations, such that the interpretation of the constraints cannot bind some meta-variables.

To overcome this drawback, we include a top-level binary construct **eval**. It is intended to split the constraints in two parts: Some constraints that can be computed before the tactic is applied and constraints that can only be evaluated after the tactic is applied. Technically, it takes as arguments two formulae, Φ_1 and Φ_2. The semantics of the construct is as follows: First $\Phi_1^{\mathcal{I},\varphi}$ is interpreted, if this results in f (in the first component), the method is not applicable, if the result is t, then the tactic is executed. The execution of the tactic part normally creates extra bindings, accumulated in ψ. The value of $\Phi_2^{\mathcal{I},\varphi\cup\psi}$ is calculated after the test application of the tactic has terminated. The method is applicable if the first component of this value is t.

In our example the constraint can be written as:

$$\mathsf{eval}(\mathsf{all}(\mathsf{prlnformula}(L_{pre})), \sim \mathsf{all}(\mathsf{prlnformula}(L_{con}))).$$

Definition 3.7: A *constraint* for a method M is either a formula in $\mathbf{WFF}^{\mathcal{M}}$, or it is of the form $\mathsf{eval}(\Phi_1, \Phi_2)$, where Φ_1 and Φ_2 are formulae in $\mathbf{WFF}^{\mathcal{M}}$.

[1] $\neg$1st(applargs(Φ_1)) is an abbreviation for application($\neg$, listcons(listfirst(applargs(Φ_1)), $\langle\rangle$)) and $\neg$2nd(applargs(Φ_1)) for application($\neg$, listcons(listfirst(listrest(applargs(Φ_1)), $\langle\rangle$))), $\langle \Phi_{21}, \Phi_{22} \rangle$ abbreviates listcons(Φ_{21}, listcons(Φ_{22}, $\langle\rangle$)).

3.2.3.3 The fixed interpretation at the meta-level

Up to now we have given the syntax and semantics of a sorted first-order language with finite quantification. The intended application of our meta-language is the specification of applicability criteria for methods. First we assume therefore a fixed model restricted to the domain of object level logical entities, namely terms, types, proof lines, proof line justifications, inference rules, subterm positions, and substitutions. Second the different categories of objects at the object level are reflected as different sorts at the meta-level to avoid unnecessary instantiations through sort restrictions, so we consider a fixed set of sorts, namely term, abstr, appl, const, var for terms and its subclasses, type for types, prln for proof lines, just for justifications of proof lines, ir for inference rules, pos for positions, and sub for substitutions. Moreover we consider finite lists of these objects. Third the functions and the predicates at the meta-level have a fixed interpretation: they are standard primitives for manipulating the objects at the object level. The restriction to a fixed interpretation for the meta-language guarantees a decidable constraint language. We elaborate on these three points successively below.

First, let us define the *fixed model*. We denote the set of terms of the object level by

$$\mathcal{D}_{term} = \mathcal{D}_{appl} \cup \mathcal{D}_{abstr} \cup \mathcal{D}_{var} \cup \mathcal{D}_{const},$$

which is divided into the set of term applications $\mathcal{D}_{appl}$, term abstractions $\mathcal{D}_{abstr}$, variables $\mathcal{D}_{var}$, and constants $\mathcal{D}_{const}$. Furthermore we have as constants the set of all types $\mathcal{D}_{type}$, proof lines $\mathcal{D}_{prln}$, justifications $\mathcal{D}_{just}$, inference rules $\mathcal{D}_{ir}$, subterm positions of the terms $\mathcal{D}_{pos}$, and substitutions on the terms $\mathcal{D}_{sub}$.

$$\mathcal{D}_{simple} = \mathcal{D}_{term} \cup \bigcup_{\mathsf{s} \in T} \mathcal{D}_{\mathsf{s}},$$

with $T = \{type, prln, just, ir, pos, sub\}$. Furthermore we have also all finite lists build upon these objects in our signature:

$$\mathcal{D}_{list} = \bigcup_{\mathsf{s} \in Types} \mathcal{D}_{list(\mathsf{s})},$$

where $\mathcal{D}_{list(\mathsf{s})}$ designates the set of all finite lists of objects in $\mathcal{D}_{\mathsf{s}}$ where $\mathsf{s} \in Types = T \cup \{term, abstr, appl, const, var\}$. All these objects together constitute our interpretation domain

$$\mathcal{D} = \mathcal{D}_{simple} \cup \mathcal{D}_{list},$$

Second we consider the *fixed set of sorts*. The division of the object level domain is directly reflected in our sort system. Therefore we define the fixed sorts as follows:

$$\mathsf{S}^{\mathcal{M}}_{term} = \{\mathsf{term}, \mathsf{appl}, \mathsf{var}, \mathsf{const}, \mathsf{abstr}\}.$$

and

$$\mathsf{S}^{\mathcal{M}}_{simple} = \mathsf{S}^{\mathcal{M}}_{term} \cup \{\mathsf{type}, \mathsf{prln}, \mathsf{just}, \mathsf{pos}, \mathsf{sub}, \mathsf{ir}\}$$

and

$$\mathsf{S}^{\mathcal{M}}_{list} = \bigcup_{\mathsf{s} \in \mathsf{S}^{\mathcal{M}}_{simple}} \{\mathsf{list}(\mathsf{s})\}$$

constituting the set of sort symbols:

$$\mathbf{SORT}^{\mathcal{M}} = \mathbf{S}^{\mathcal{M}}_{simple} \cup \mathbf{S}^{\mathcal{M}}_{list}.$$

As mentioned in definition 3.5, the interpretation of a term $t \in \mathbf{T}^{\mathcal{M}}_{\mathsf{s}}$ is an element of $\mathcal{D}_{\mathsf{s}}$ for all $\mathsf{s} \in Types$. Objects of the sorts prln, just, and ir stand for proof lines, justifications, and inference rules of the ND calculus, respectively. Since terms are further subdivided into applications, abstractions, variables, and constants, the sort term, has the corresponding subsorts appl, abstr, var, and const.

Third the use of a *fixed set of functions and predicates* can be motivated by the intended application of the meta-language: We want to make statements about the underlying object language, in our case, ND proofs with $\mathcal{POST}$ formulae. Therefore we choose a set of selectors and constructors for the different objects like proofs, inference rules, proof-lines, types, terms, substitutions, and lists of these objects. Our language and the whole approach is not restricted to the following functions as long as the interpretation of a formula is always decidable.

We use the following functions and relations with a fixed interpretation:

For proofs, proof lines, justifications, etc.: The actual proof state in our planning process is stored in cprf, a constant that has the function declaration cprf⩽list(prln).

We can also manipulate proof lines. A proof line l has the form $(\alpha)\quad H \vdash F\quad J$ and the justification J has the form $r(l')$, where H, l' are lists of proof lines, F is a formula, J is a justification and r is an inference rule (cf. section 2.2). The following accessors are given: $\mathsf{prlnhyps}^{\mathcal{I}}(l) = H$ with the function declaration prlnhyps⩽prln → list(prln), $\mathsf{prlnformula}^{\mathcal{I}}(l) = F$ having the function declaration prlnformula⩽prln → term, $\mathsf{prlnjust}^{\mathcal{I}}(l) = J$ with the function declaration prlnjust⩽prln → just, $\mathsf{justrule}^{\mathcal{I}}(J) = r$ with the function declaration justrule⩽just → ir, and $\mathsf{justlines}^{\mathcal{I}}(J) = l'$ with the function declaration justlines⩽just → list(prln).

For terms: Furthermore we have primitives for manipulating terms:

$\mathsf{newconst}^{\mathcal{I}}(\tau)$ with the function declaration newconst⩽type → const gives a new constant of type τ and $\mathsf{newvar}^{\mathcal{I}}(\tau)$ with the function declaration newvar⩽type → var gives a new variable of type τ. We get the type of a term with termtype⩽term → type.

$\mathsf{termoccs}^{\mathcal{I}}(t_1, t_2)$ with function declaration termoccs⩽term × term → list(pos) gives a list of positions of occurrences of term t_1 in t_2. We can get all occurences of variables in a term by the accessor termvars⩽term → list(pos). $\mathsf{termatpos}^{\mathcal{I}}(t, p)$ with the function declaration termatpos⩽term × pos → term gives the subterm at position p in the term t. For instance, $\mathsf{termatpos}(f(g(x, y), z), [1\ 1]) = x$. termrploccs⩽term × term × term → term has the following interpretation: $\mathsf{termrploccs}^{\mathcal{I}}(t_1, t_2, t_3)$ replaces all free occurences of t_1 in t_2 with the term t_3.

The most general unifier and the most general matcher for two terms can be computed with the functions termmgu⩽term×term → subst and termmgm⩽term×term → subst.

We also have predicates atom,impl,neg,conj,disj,eqv,all,ex⩽term, with the interpretations:

$\mathsf{atom}^{\mathcal{I}}(F) = \mathsf{t}$	iff	F is an atom.		$\mathsf{eqv}^{\mathcal{I}}(F) = \mathsf{t}$	iff	F is an equivalence.
$\mathsf{impl}^{\mathcal{I}}(F) = \mathsf{t}$	iff	F is an implication.		$\mathsf{all}^{\mathcal{I}}(F) = \mathsf{t}$	iff	F is a universally
$\mathsf{neg}^{\mathcal{I}}(F) = \mathsf{t}$	iff	F is a negation.				quantified formula.
$\mathsf{conj}^{\mathcal{I}}(F) = \mathsf{t}$	iff	F is a conjunction.		$\mathsf{ex}^{\mathcal{I}}(F) = \mathsf{t}$	iff	F is an existentially
$\mathsf{disj}^{\mathcal{I}}(F) = \mathsf{t}$	iff	F is a disjunction.				quantified formula.

For applications: As a special case of terms, we can create an application $t(t_1, \ldots, t_n)$ using $\mathsf{application}^{\mathcal{I}}(t, l)$ with the function declaration $\mathsf{application} \leqslant \mathsf{term} \times \mathsf{list(term)} \rightarrow \mathsf{appl}$ where $l = \langle t_1, \ldots, t_n \rangle$ is a list of terms. For an application we can project on the function using $\mathsf{applfunc} \leqslant \mathsf{appl} \rightarrow \mathsf{term}$ or on the arguments using $\mathsf{applargs} \leqslant \mathsf{appl} \rightarrow \mathsf{list(term)}$ such that $\mathsf{applfunc}^{\mathcal{I}}(t(t_1, \ldots, t_n)) = t$, and $\mathsf{applargs}^{\mathcal{I}}(t(t_1, \ldots, t_n)) = \langle t_1, \ldots, t_n \rangle$.

For abstractions: It is also possible to create an abstraction with $\mathsf{abstraction} \leqslant \mathsf{var} \times \mathsf{term} \rightarrow \mathsf{abstr}$ with the following arguments: $\mathsf{abstraction}^{\mathcal{I}}(x, f) = \lambda x.f$. The selectors for abstractions are: $\mathsf{abstrscope}$ with the function declaration $\mathsf{abstrscope} \leqslant \mathsf{abstr} \rightarrow \mathsf{term}$ giving the scope of the abstraction, i.e., $\mathsf{abstrscope}^{\mathcal{I}}(\lambda x.f) = f$, and $\mathsf{abstrvar}$ with the function declaration $\mathsf{abstrvar} \leqslant \mathsf{abstr} \rightarrow \mathsf{var}$ projecting on the abstracted variable of an abstraction, $\mathsf{abstrvar}^{\mathcal{I}}(\lambda x.f) = x$. The basic rule of beta-reduction is also incorporated as a function $\mathsf{betareduce} \leqslant \mathsf{abstr} \times \mathsf{term} \rightarrow \mathsf{term}$ with the interpretation: $\mathsf{betareduce}^{\mathcal{I}}(\lambda x.f, g) = \{x \mapsto g\}f$. The notation $\{x \mapsto g\}f$ means that all occurences of x in f are replaced by g. A list of all variables in a term bound by abstractions can be computed using $\mathsf{abstrboundvars} \leqslant \mathsf{term} \rightarrow \mathsf{list(var)}$.

For positions: The subterms can be selected by using positions, also known as occurrences. The positions are constructed by the empty position $[] \leqslant \mathsf{pos}$ or recursively by $\mathsf{poscons} \leqslant \mathsf{nat} \times \mathsf{pos} \rightarrow \mathsf{pos}$.[2] The two selectors are $\mathsf{posfirst} \leqslant \mathsf{pos} \rightarrow \mathsf{nat}, \mathsf{posrest} \leqslant \mathsf{pos} \rightarrow \mathsf{pos}$. Furthermore we have a function $\mathsf{posappend} \leqslant \mathsf{pos} \times \mathsf{pos} \rightarrow \mathsf{pos}$ with the following interpretation: Suppose that $p = [n_1 n_2 \ldots n_i]$ and $p' = [n'_1 n'_2 \ldots n'_k]$, then: $\mathsf{posfirst}^{\mathcal{I}}(p) = n_1$, and $\mathsf{posrest}^{\mathcal{I}}(p) = [n_2 \ldots n_i]$ and $\mathsf{posappend}^{\mathcal{I}}(p, p') = [n_1 n_2 \ldots n_i\ n'_1 n'_2 \ldots n'_k]$,

For types: Types can be recursively generated using $\mathsf{itype}, \mathsf{otype} \leqslant \mathsf{type}$ interpreted as types ι and o. Complex types can be generated using $\mathsf{typeabstr} \leqslant \mathsf{list(type)} \times \mathsf{type} \rightarrow \mathsf{type}$ interpreted as:

$$\mathsf{typeabstr}^{\mathcal{I}}(\langle \tau_1, \ldots, \tau_n \rangle, \tau) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau.$$

The selectors for types are $\mathsf{typedomain}, \mathsf{typerange} \leqslant \mathsf{type} \rightarrow \mathsf{type}$ with the interpretation

$$\mathsf{typedomain}^{\mathcal{I}}(\tau_1 \times \cdots \times \tau_n \rightarrow \tau) = \langle \tau_1, \ldots, \tau_n \rangle,$$

and

$$\mathsf{typerange}^{\mathcal{I}}(\tau_1 \times \cdots \times \tau_n \rightarrow \tau) = \tau,$$

For substitutions: Substitutions have as primitives $\{\} \leqslant \mathsf{subst}$ interpreted as the empty substitution, $\mathsf{substcompose} \leqslant \mathsf{subst} \times \mathsf{subst} \rightarrow \mathsf{subst}$ interpreted as the composition

[2] nat is the standard sort for natural numbers, which is only used for the positions.

of two substitutions, substaddcomp⩽var × term × subst → subst with the following interpretation:

$$\mathsf{substaddcomp}^{\mathcal{I}}(x, t, \sigma) = \{x \mapsto t\} \cup \sigma,$$

adding a new component to a substitution. Furthermore we can restrict a substitution in its domain by substrestrict⩽subst × list(var) → subst with the meaning:

$$\mathsf{substrestrict}^{\mathcal{I}}(\sigma, l) = \sigma\,|_{\{x_1,\ldots,x_n\} \cap \mathrm{Dom}(\sigma)},$$

where $l = \langle x_1, \ldots, x_n \rangle$. With the predicate substdomain⩽var × subst we can check whether a variable is in the domain of a substitution and with substcodomain⩽term × subst whether a term is in the codomain of a substitution. The function substapply⩽ subst × term → term applies a substitution to a term by simultaneously replacing the domain variables in the term with the corresponding terms of the codomain.

For lists: Lists can be constructed recursively with ⟨⟩⩽list(s) interpreted as the empty list[3] and listcons⩽s × list(s) → list(s) interpreted as adding an element of sort s at the front of a list, and listappend⩽list(s) × list(s) → list(s) interpreted as appending two lists. The selectors are as usual listfirst⩽list(s) → s interpreted as giving the first element of a given list, and listrest⩽list(s) → list(s) interpreted as giving the list without the first element of a given list.

3.2.4 Decidability of the Constraint Language

A main criterion for the method language is its tractability, which strongly depends on the decidability of the constraint language. When we have the special case of an eval–constraint, the termination can not be guaranteed because there can be nonterminating procedural program code in the procedural part of a tactic. Such tactics should be checked for applicability only with great care and with the help of additional mechanisms, for instance the application of such a procedure should be resource bounded. Since we assume that these constraints occur very rarely, we restrict our attention to the constraints without an eval and prove the decidability of the resulting constraint language.

Theorem (Decidability): *If all variables of the constraints are bound by matching the premises and conclusions against the proof lines occuring in the actual proof state, then the interpretation of the constraint terminates with* $\langle \mathsf{t}, \varphi \rangle$ *or* $\langle \mathsf{f}, \varphi \rangle$ *as value, where* φ *is a partial assignment function.*

Proof: First we prove the decidability for the atomic formulae. When the constraint is an atom of the form $p(t_1, \ldots, t_n)$, then we have a fixed interpretation for $p^{\mathcal{I}}$ and since the t_i, $1 \leq i \leq n$, are computable, we can decide whether the relation $p^{\mathcal{I}}$ holds or not.

For compound formulae with a negation, a conjunction and a disjunction as connective, we can also decide if the formula holds or not, by induction.

[3] We assume that we have different constants of the empty list of the different sorts. In the following we denote all these different constants with the same symbol.

For quantified formulae, the quantifiers range only over finite lists. Therefore they are decidable, since their evaluation is computationally similar to finite conjunctions or finite disjunctions, respectively. ■

3.2.5 Homomorphy Example

We want to illustrate our method language by an example of a method for proving theorems in the field of algebra. Its proof strategy can be informally 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 to show $P(f(c))$.* The very idea is that f is a homomorphism for the property P and that f can be "rippled away" (compare [6]).

Method : `hom1-1`	
Declarations	$L_1, L_2, L_3, L_4, L_5, L_6$:prln H_1, H_2, H_6:list(prln) J_1, J_2, J_3:just X, Y:var P, F, C:const $\Phi, \Psi, \Psi_1, \Psi_2$:term
Premises	$L_1, L_2, \oplus L_3$
Constraint	$\bigwedge^{H_1} x$:prln.$\bigvee^{H_6} y$:prln.$x = y$ & $\bigwedge^{H_2} x$:prln.$\bigvee^{H_6} y$:prln.$x = y$ & $\sim$ termoccs$(F, \Phi) \doteq \langle\rangle$ & termtype$(C) \doteq$typerange(termtype(F)) & $\Psi_1 \leftarrow$ termrploccs(X, Ψ, C) & $\Psi_2 \leftarrow$ termrploccs$(X, \Psi, F(C))$
Conclusions	$\ominus L_6$
Declarative Content	(L_1) $H_1 \vdash \forall Y.\Phi$ (J_1) (L_2) $H_2 \vdash \forall X.P(X) \leftrightarrow \Psi$ (J_2) (L_3) $H_6 \vdash P(C)$ (J_3) (L_4) $H_6 \vdash \Psi_1$ (`def-e` L_2, L_3) (L_5) $H_6 \vdash \Psi_2$ (OPENL_1, L_4) (L_6) $H_6 \vdash P(F(C))$ (`def-i` L_2, L_5)
Procedural Content	`schema – interpreter`

Figure 3.3: The `hom1-1` method.

Suppose we want to prove the theorem that the converse relation of a binary relation ρ is symmetric (formally: *symmetric*(*converse*(ρ))). The method `hom1-1` can be applied by substituting *converse*, *symmetric*, and ρ for the meta-variables F, P, and C, respectively[4]. The method `hom1-1` proposes *symmetric*(ρ) as a new line which can be used to prove

[4]In the following the capital letters denote meta-variables, while the object level elements are written in lowercase letters.

symmetric(*converse*(ρ)) together with the definitions of *symmetric* and *converse*. The details of using this method in proof planning are discussed in the next chapter.

The constraint of the method states the following: The first two lines express that H_1 and H_2, the hypotheses of line L_1 and L_2, are subsets of H_6, the hypotheses of Line L_6. The third line states that F must occur in Φ, the fourth line means that certain types must be equal (otherwise the newly created formula $P(C)$ is not well-typed). The fifth line means that Ψ_1 is created by replacing all free occurences of X in Ψ by C, and the sixth line that Ψ_2 is obtained by replacing all occurences of X in Ψ by $F(C)$.

Chapter 4

Planning

> We secure our mathematical knowledge by *demonstrative reasoning* , but we support our conjectures by *plausible reasoning* ... Demonstrative reasoning is safe, beyond controversy, and final. Plausible reasoning is hazardous, controversial, and provisional.
>
> *George Pólya, Mathematics and Plausible Reasoning*

In this chapter we specify our proof planning mechanism and the semantics of our methods from the planning perspective. After giving a motivation and showing an example, we formally define STRIPS-like plan operators from our methods and then show their use in a planning algorithm.

4.1 Motivation

To understand the proof planning process, please remember that the goal of proof planning is to fill gaps in a given partial proof tree by forward and backward reasoning. As a first attempt we adopt a STRIPS-like planning paradigm, where the plan 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 *plan operators* in order to find a complete *plan* (that is a sequence of instantiated plan operators) from a given *initial state* to a *terminal state.*

Concretely a *planning state* contains 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 as premises to bridge it). Thus on the meta-level of planning there are two labels: "(?)" states that the argument is an open line and "(!)" states that the argument is a support line. The initial planning state consists of all lines in the initial problem; the assumptions are the support lines and the conclusion is the only open line. The terminal planning state is reached when there is no more open line in the planning state.

Aimed at modeling a human-oriented reasoning, we want our proof planning mechanism to be able to perform both forward reasoning and backward reasoning. 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 backward reasoning the methods are applied from their conclusions to their premises, that is, some premises become new open lines in the planning state and the open lines matched with the conclusions of the method are deleted in the planning state because they are proved by the method. In forward reasoning the methods are applied from their premises to their conclusions, that is, some conclusions become new support lines in the planning state. In order to achieve this bi-directional reasoning direction with a uni-directional planning mechanism, the planning direction must be independent from the reasoning direction. For this reason we included the labels "$\oplus$" and "$\ominus$" in the premises and conclusions slot of a method.

The labels can be examined from another perspective, namely the relation between the static proof represented in a method and the dynamic proof planning process. The specification of methods without the labels gives only a static (viewed from the completed proof) description 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 (we will call them *required* lines) have to be present in the planning state for the method to be applicable, and which are constructed by the method. We do this by marking the lines which are constructed by a method and which will be inserted into the planning state with the label "$\oplus$". Additionally it is useful to specify that some of the required lines of a method should not be used again by the planner. This allows to keep the planning state small and thus to optimize the search for plans. We 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 of reasoning (forward vs. backward) of the method by specifying if the method is applied from its premises to its conclusions or vice versa.

In order to demonstrate how the methods are used in the planning process, we show how the corresponding STRIPS plan operators would look like. We presume the STRIPS mechanism [11] as prerequisite and only mention that a STRIPS plan operator consists of three slots: the *preconditions* slot, the *delete* slot, and the *add* slot. A STRIPS plan operator is applicable in a planning state if the propositions in the preconditions slot are present in the planning state. When a method is applied to a planning state, the propositions in the delete slot are removed from the planning state and the propositions of the add slot are inserted.

4.2 An Example

Before continuing in describing how a STRIPS plan operator is constructed from a method, we give an example that illustrates the use of the labels. Figure 4.1 shows the specification of two simple methods[1], as well as the STRIPS plan operators defined by them. These methods are only a simplified version of a more general class of methods applying assertions (definitions and theorems). For a comprehensive study of this class which approximates basic proof steps encountered in informal mathematical practice see [16].

[1]Note that these methods are not exactly in the syntax defined in chapter 3. They are simplified here for didactic reason. In particular $\overline{X}$ designates a tuple of objects and $\forall\overline{X}.P(\overline{X})$ designates $\forall X_1.\ldots\forall X_n.P(X_1,\ldots,X_n)$. The constraint slot is empty and the declarations slot is obvious, hence these two are left out.

Method `def-i` (Definition Introduction)		
Premises	`Definition:`	$\forall \overline{X}.P(\overline{X}) \leftrightarrow \Psi_{\overline{X}}$
	$\oplus$`expanded-line:`	$\Psi_{\overline{T}}$
Conclusions	$\ominus$`defined-line:`	$P(\overline{T})$

$\Longrightarrow$

Plan-Op `def-i`

Pre:	(!)	`Definition`
	(?)	`defined-line`
Del:	(?)	`defined-line`
Add:	(?)	`expanded-line`

Method `def-e` (Definition Elimination)		
Premises	`Definition:`	$\forall \overline{X}.P(\overline{X}) \leftrightarrow \Psi_{\overline{X}}$
	$\ominus$`defined-line:`	$P(\overline{T})$
Conclusions	$\oplus$`expanded-line:`	$\Psi_{\overline{T}}$

$\Longrightarrow$

Plan-Op `def-e`

Pre:	(!)	`Definition`
	(!)	`defined-line`
Del:	(!)	`defined-line`
Add:	(!)	`expanded-line`

Figure 4.1: The specification of the methods `def-i` and `def-e` and the corresponding plan operators. The terms 'introduction' and 'elimination' refer to the static proof; in the dynamic proof planning process the definition is expanded in both methods.

The method `def-i` applies a definition for the predicate P to an open line (marked with "(?)" in the plan operators and in the planning state) and constructs a new open line with the expanded definition (by backward reasoning); `def-e` applies a definition to a support line (marked with "(!)") and constructs a new support line (by forward reasoning). In order to achieve this dynamic behavior the labels are used for the lines in the specifications as follows.

It is obvious that in both methods the line of the definition is required because it is not reasonable to "guess" a definition[2]; furthermore the definition line must not be deleted in the planning state since it might be used more than once in a proof. Therefore `Definition` has no label in our example; in the STRIPS plan operators this line must occur in the precondition slot as a support line. Clearly in both methods `defined-line` must be a required line and in `def-i` (and analogously in `def-e`) it is useless after the application of the definition because an open line needs to be proved only once. It is labeled with "$\ominus$" and must occur in the preconditions slot as well as in the delete slot of the STRIPS plan operators. Since `expanded-line` is constructed by the methods, it is labeled with "$\oplus$" and occurs in the add slot of the STRIPS plan operators.

When applying `def-i`, `defined-line` has to be matched with an open line and the newly constructed `expanded-line` becomes an open line. But when applying `def-e`, `defined-line` must be matched with a support line and `expanded-line` becomes a new support line. Therefore in the STRIPS plan operator constructed from `def-i`, namely `defined- line` and `expanded-line`, occur as open lines. Analogously in the STRIPS plan operator constructed from `def-e` these lines occur as support lines. Figure 4.2 shows an example of the effect of the methods `def-e` and `def-i` on the planning state. Line 1 is the definition of the predicate *symmetric*. Line 3 is constructed by applying `def-e` to line 1 and 2, that is, by expanding the definition of *symmetric* in forward reasoning; line 8 is constructed by applying `def-i` to line 1 and 9, that is, by expanding the definition of *symmetric* in backward reasoning.

[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 distinguish it from simply applying a definition.

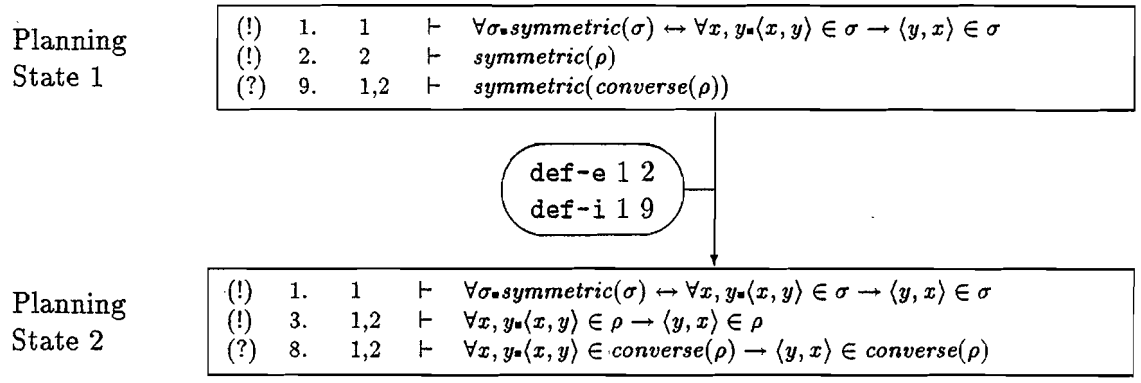


Figure 4.2: Using `def-i` and `def-e` in the planning process

4.3 Defining STRIPS Plan Operators from Methods

Now we formally define how the specification of a method corresponds to a STRIPS plan operator. The required slots of the specification are the premises, the constraint, and the conclusions. The premises and conclusions contain lines that are labeled with "⊕" or "⊖" or are unlabeled; the constraint contains an additional logical statement that can be evaluated to true or false. With this information we define a STRIPS plan operator that has three slots: the precondition list, the delete list, and the add list.

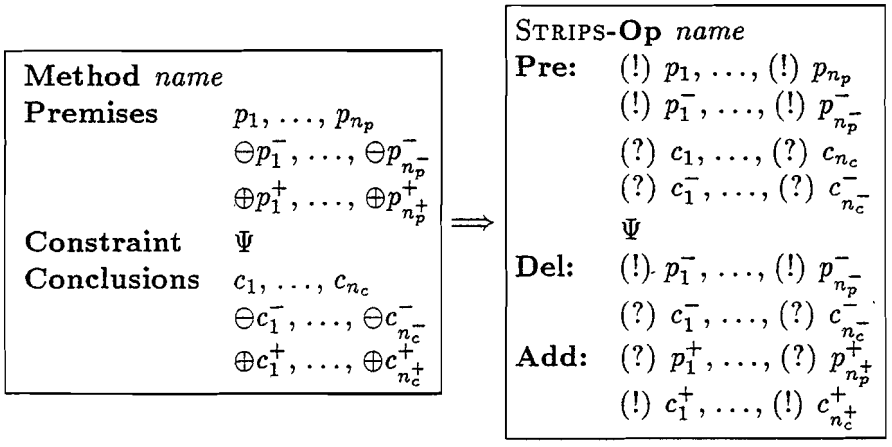


Figure 4.3: Defining a plan operator from a method's specification

Figure 4.3 shows the generation of the STRIPS plan operator from a method. It defines the most general case, although usually not all possible labels are used in a single method. The unlabeled lines in a method go to the preconditions slot and the lines with a "⊕" to the add slot. The lines labeled with "⊖" are moved to the precondition slot and to the delete slot. In the preconditions and delete slot the premises become support lines and the conclusions become open lines. In the add slot it is the other way round. The content of the constraint slot is inserted into the preconditions slot of a STRIPS operator. Note that it is not a proof line that is present in the planning state, but a formula specifying

While there are still open lines in the planning state

1. Find all possibilities of an application of the methods
   (a) Select a method M.
   (b) Find all possibilities of matching the lines in M's precondition slot with lines in the planning state.
   (c) Evaluate the *constraint* of M with the bindings of the last step.
2. Select the "best" method $\widetilde{M}$ (this is the choice point for the backtracking mechanism and the point where some heuristic control can take place)
3. Apply the plan operator to the planning state
   (a) Insert the lines in the add slot of $\widetilde{M}$ into the planning state.
   (b) Remove the lines in the delete slot of $\widetilde{M}$ from the planning state.

Figure 4.4: The planning algorithm

an additional applicability condition. From now on the preconditions, delete, and add slot refer to the corresponding slot in a STRIPS plan operator defined by the method.

In figure 4.4 we give an abstract view of the planning algorithm. The plan operators are applied as usual. Note that backward planning is not possible since the terminal state is defined by the absence of OPEN lines. During the matching of the lines in the preconditions slot and the evaluation of the constraint all meta-variables should have been bound to object level terms. Therefore the new lines of step 3.(a) can be constructed by simply instantiating the meta-variables.

Once a complete proof plan is found, all methods (i.e. their tactics) in the proof plan are successively applied to construct a calculus level proof. 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 the `homl-1` method, for instance) that define further gaps awaiting to be closed by the proof planner. Therefore the verification phase which follows the application of the methods, may result in a recursive call to the planner or in backtracking. While a recursive call refines a plan and models hierarchical planning, the backtracking rejects the plan and calls the proof planner in order to find a different plan.

4.3.1 Homomorphy Example (Continued)

Having illustrated the basic framework, let us examine a slightly more complex example, related to the `homl-1` method. The method is shown in figure 3.3. Note that line L_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) and results in a hierarchical proof planning: since line L_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 `homl-1`. This will result in a recursive

call of the planner in the following verification phase.

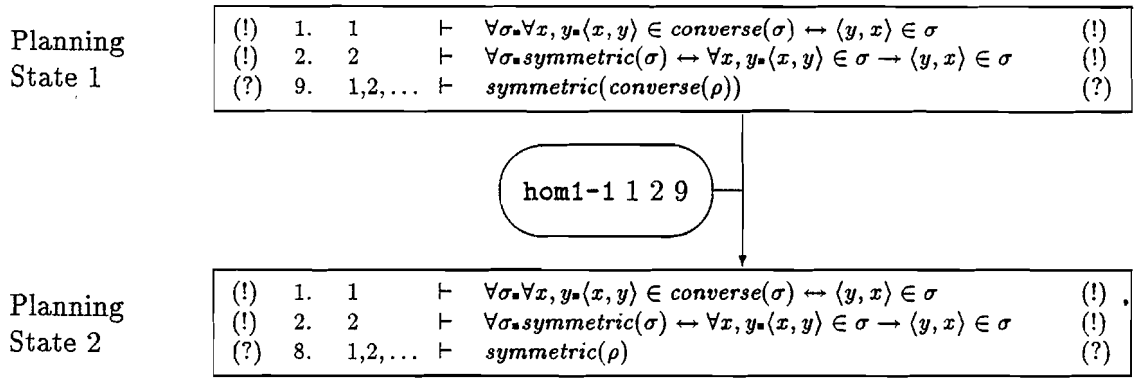


Figure 4.5: Using `hom1-1` in the planning process

For example, to prove that the converse relation of a binary relation ρ is symmetric (formally: $symmetric(converse(\rho))$), the method `hom1-1` can be applied by instantiating the meta-variables F, P, and C by *converse*, *symmetric*, and ρ, respectively. While in figure 4.2 we filled the gap between $symmetric(\rho)$ and $symmetric(converse(\rho))$ which were both existing lines, in this example the method `hom1-1` proposes $symmetric(\rho)$ as a new line which can be used to prove $symmetric(converse(\rho))$ together with the definitions of *symmetric* and *converse*. Figure 4.5 shows the transition of the planning state. If the plan can be completed (by proving the remaining OPEN line with some additional information about ρ) the proof resulting from the application of the tactic `hom1-1` would look like in figure 4.6.

1.	1	⊢	$\forall\sigma\forall x,y\langle x,y\rangle \in converse(\sigma) \leftrightarrow \langle y,x\rangle \in \sigma$	(J1)
2.	2	⊢	$\forall\sigma\, symmetric(\sigma) \leftrightarrow \forall x,y\langle x,y\rangle \in \sigma \rightarrow \langle y,x\rangle \in \sigma$	(J2)
3.	1,2,...	⊢	$symmetric(\rho)$	(J3)
4.	1,2,...	⊢	$\forall x,y\langle x,y\rangle \in \rho \rightarrow \langle y,x\rangle \in \rho$	(`def-e` 3 2)
5.	1,2,...	⊢	$\forall x,y\langle x,y\rangle \in converse(\rho) \rightarrow \langle y,x\rangle \in converse(\rho)$	(OPEN 1 4)
6.	1,2,...	⊢	$symmetric(converse(\rho))$	(`def-i` 2 5)

Figure 4.6: 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 via matching when applying the plan operator of `hom1-1`, J3 will be inserted by the rest of the proof plan. The justifications of lines 4 and 6 stand for the subproofs generated by the application of the tactics of these methods, whereas the justification of line 5 defines a new gap with support lines containing lines 1 and 4.

Chapter 5

Extending the Reasoning Repertoire by Meta-Methods

> ... et chaque vérité que je trouvais étant une règle qui me servait après à en trouver d'autres, ...
>
> *René Descartes, Discours de la Méthode*

It is one of the main features contributing to the problem solving competence of mathematicians that they can extend their current problem solving methods by adapting them to suit new situations (see [23] for mathematical reasoning and [27] for general problem solving). Newell has pointed out in [21] that the mechanization of this procedure is hard but essential in order to go beyond the reasoning power of existing systems.

In the previous chapters of this paper, a declarative approach for formulating methods has been introduced in order to mechanize aspects of this procedure by so-called meta-methods. In this chapter we discuss the notion of a meta-method and briefly illustrate the planning process with meta-methods with the help of an example. A more detailed discussion can be found in [18].

By discerning the declarative part of tactics, it is now feasible to formulate meta-methods adapting the declarative part of existing methods and thus come up with novel methods. In a framework where tactics consist only of procedural knowledge, we would in effect be confronted with the much more difficult problem of adapting procedures in order to achieve the above.

We define a *meta-method* as consisting of:

- A *body*: a procedure which takes as input a method, and possibly further parameters from the planner (in particular the current state of proof planning) and generates a new method with the same procedural part.
- 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[1].

[1] 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	
Declarations	$L_1, L_2, L_3, L_4, L_5, L_6, L_7, L_8$:prln H_1, H_2, H_8:list(prln) J_1, J_2, J_3, J_4:just X, Y:var P, F', C, D:const $\Phi, \Psi, \Psi_1, \Psi_1', \Psi_2$:term
Premises	$L_1, L_2, \oplus L_3, \oplus L_4$
Constraint	$\bigwedge^{H_1} x$:prln.$\bigvee^{H_8} y$:prln.$x = y$ & $\bigwedge^{H_2} x$:prln.$\bigvee^{H_8} y$:prln.$x = y$ & $\sim$ termoccs$(F', \Phi) \doteq \langle\rangle$ & termtype$(C) \doteq$typerange(termtype(F')) & termtype$(D) \doteq$typerange(termtype(F')) & $\Psi_1 \leftarrow$ termrploccs(X, Ψ, C) & $\Psi_1' \leftarrow$ termrploccs(X, Ψ, D) & $\Psi_2 \leftarrow$ termrploccs$(X, \Psi, F'(C, D))$
Conclusions	$\ominus L_8$
Declarative Content	(L_1) $H_1 \vdash \forall Y.\Phi$ (J_1) (L_2) $H_2 \vdash \forall X.P(X) \leftrightarrow \Psi$ (J_2) (L_3) $H_8 \vdash P(C)$ (J_3) (L_4) $H_8 \vdash P(D)$ (J_4) (L_5) $H_8 \vdash \Psi_1$ (def-e L_2, L_3) (L_6) $H_8 \vdash \Psi_1'$ (def-e L_2, L_4) (L_7) $H_8 \vdash \Psi_2$ (OPEN L_1, L_5, L_6) (L_8) $H_8 \vdash P(F'(C, D))$ (def-i L_2, L_7)
Procedural Content	schema – interpreter

Figure 5.1: The hom1-2 method.

We illustrate this definition with the hom1-1 method introduced in section 3.2.5. The method hom1-1 simplifies a problem by generating an intermediate goal, where a *unary* function symbol is eliminated. Suppose we are facing the new but similar problem of proving that the intersection of symmetric relations is itself a symmetric relation. What we need is a variant of hom1-1, which is able to handle a *binary* function symbol (*intersection*) in a similar way.

In the following, we illustrate how to use a meta-method called add-argument to obtain a binary version hom1-2 from the unary version hom1-1. hom1-1 is suited to situations with a unary predicate constant P and a unary function constant F, while hom1-2 can handle situations with a unary predicate constant P and a binary function constant F'. Note that P, F, and F' 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.

This meta-method is supposed to add an argument to a key constant symbol F which is

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

a unary predicate or function used in a method, the modified function or predicate constant is called F'.

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

- replace all occurrences of terms $\mathtt{F}(x)$ by $\mathtt{F}'(x, y)$ and augment the corresponding quantifications,
- replace all occurrences of terms $\mathtt{F}(C)$ by $\mathtt{F}'(C, D)$ (D has to be a new meta-variable standing for a constant),
- if C occurs in a proof line, but not in a term $\mathtt{F}(C)$, a copy of this line will be inserted into the proof schema, replacing C by D (in the example below, line 4 is copied from 3). Such a copy must be accompanied by a corresponding augmentation to the specification of the method.

Let us reiterate that as the crucial advantage of separating the procedural and the declarative knowledge in methods, the procedural part of M can be taken over for the new method. The method `hom1-2` below can be obtained by applying `add-argument` with the arguments `hom1-1` and F.

Note that the `hom1-2` method is indeed useful to solve the intended problem of showing that the intersection of two symmetric relations is symmetric too. From the initial problem the method `hom1-2` produces the following partial proof:

1.	1	$\vdash \forall\rho,\sigma.\forall x,y.\langle x,y\rangle \in \mathit{intersection}(\rho,\sigma) \leftrightarrow \langle x,y\rangle \in \rho \wedge \langle x,y\rangle \in \sigma$	(J1)
2.	2	$\vdash \forall\sigma.\mathit{symmetric}(\sigma) \leftrightarrow \forall x,y.\langle x,y\rangle \in \sigma \rightarrow \langle y,x\rangle \in \sigma$	(J2)
3.	1,2,...	$\vdash \mathit{symmetric}(\rho)$	(J3)
4.	1,2,...	$\vdash \mathit{symmetric}(\sigma)$	(J4)
5.	1,2,...	$\vdash \forall x,y.\langle x,y\rangle \in \rho \rightarrow \langle y,x\rangle \in \rho$	(`def-e` 2 3)
6.	1,2,...	$\vdash \forall x,y.\langle x,y\rangle \in \sigma \rightarrow \langle y,x\rangle \in \sigma$	(`def-e` 2 4)
7.	1,2,...	$\vdash \forall x,y.\langle x,y\rangle \in \mathit{intersection}(\rho,\sigma) \rightarrow \langle y,x\rangle \in \mathit{intersection}(\rho,\sigma)$	(OPEN 1 5 6)
8.	1,2,...	$\vdash \mathit{symmetric}(\mathit{intersection}(\rho,\sigma))$	(`def-i` 2 7)

Analogously a method `hom2-1` (for handling a unary function symbol but a binary predicate symbol) can be obtained by applying `add-argument` with the arguments `hom1-1` and P.

In an interactive proof development environment like Ω-MKRP [17] the user has the opportunity to choose and apply a meta-method himself. To provide the user with heuristic support or even to automatize this complex procedure, heuristics are necessary. For discussions on heuristics and for a preliminary classification of meta-methods, readers are referred to [18].

Chapter 6

Conclusion

A good mathematician has to learn a remarkable repertoire of technical knowledge. On the one hand this is factual knowledge, namely definitions, theorems, and proofs. On the other hand he has to learn problem solving know-how as well. This kind of knowledge consists of standard *methods* for manipulating proofs, like mathematical induction or diagonalization. Among other important activities of mathematical reasoning (like defining new concepts and adapting a definition so that new theorems can be proved) there is one very important feature, namely the ability to adapt existing problem solving facilities to new, not directly fitting situations.

In order to model such mechanical modification, we have presented in this report a formal definition of a method language. In particular we have defined the notion of a method with the components: declarations, premises, constraints, conclusions, declarative content, and procedural content. The main feature is the separation of procedural and declarative knowledge in the tactic part. In this way, all parts are declarative and subject to automatic modification, except the procedural content of the tactic. Another emphasis of this paper is a detailed description of a declarative constraint language. With this language we can bind free variables and formulate applicability conditions not expressible in terms of proof line schemata. In order to compromise competing requirements, namely expressivity, adaptability, and tractability, we have designed the constraint language as a decidable variant of sorted first-order logic.

The duality of the semantics of a method corresponds to its two main aspects: a method is a tactic and a plan operator. The semantics of the tactic part specifies the effect of its execution while the semantics of the plan operator specifies the behavior in a planning process. In order to model the latter we have presented how a method can be translated into a STRIPS plan operator, together with a first version of a planning algorithm. In the last chapter we have shown how the modification of methods can be performed by so-called meta-methods, producing new methods applicable in new situations.

To summarize we have proposed a declarative extension to Bundy's proof planning framework in order to enable reformulations of methods. Much work remains to be done. We are currently extending our first implementation, in particular an interpreter for the constraint language is under construction. The efficiency of the whole approach depends strongly on the implementation of a planning algorithm more suitable than the naive STRIPS-like planner. The ultimate adequacy of our approach, of course, still has to be judged by experience accumulated with more examples.

Acknowledgement

We would like to thank Michael Kohlhase for many fruitful discussions and his contributions to related joint works. Thanks are also due to Erica Melis for many inspiring discussions and to Ralf Sesseler who implemented the prototype. In an early stage of this work the first two authors had many clarifying sessions with Jörg Denzinger and Inger Sonntag, we would like to thank them as well.

Appendix A

Examples

The framework described in this paper is currently under implementation. In this appendix we want to show some results of a prototype implementation of the proof planning algorithm. Only the specification part of methods has been implemented. The tactic part, which is responsible for the application of methods and for the construction of a proof, is still missing. Therefore the methods presented here contain only information used for the planning process, that is, there are no additional lines in the declarative content that would only be used when the tactic is applied and the proof is constructed.

A.1 Methods

The following methods are used in the examples. The first method is the `hom1-2`-Method that has been presented before.

Method : `hom1-2`	
Declarations	L_1, L_2, L_3, L_4, L_5 :prln; H_1, H_2, H_5 :list(prln); J_1, J_2, J_3, J_4 :just; P, F', C, D :const;
Premises	$L_1, L_2, \oplus L_3, \oplus L_4$
Constraint	$\bigwedge^{H_1} x$:prln.$\bigvee^{H_5} y$:prln.$x = y$ & $\bigwedge^{H_2} x$:prln.$\bigvee^{H_5} y$:prln.$x = y$
Conclusions	$\ominus L_5$
Declarative Content	L_1 $H_1 \vdash \forall Y.\Phi$ (J_1) L_2 $H_2 \vdash \forall X.P(X) \leftrightarrow \Psi$ (J_2) L_3 $H_5 \vdash P(C)$ (J_3) L_4 $H_5 \vdash P(D)$ (J_4) L_5 $H_5 \vdash P(F'(C,D))$ (`hom1-2`)
Procedural Content	`schema – interpreter`

The next five methods are reflecting simple ND calculus rules. The distribution of the labelled lines in the premises and conclusion slot of the `implies-I`-method seems to be a little bit weird. This stems from the introduction of a hypothesis which can hardly be

described in terms of premises and conclusions. But the planning operator defined by this method performs what it is supposed to do: In a planning state an open line $H \vdash \Phi \rightarrow \Psi$ can be replaced by a support line $(L_1) \vdash \Phi$ and an open line $H \cup \{L_1\} \vdash \Psi$.

Method : implies-I	
Declarations	L_1, L_2, L_3 :prln; H_2, H_3 :list(prln); J :just; Ψ, Φ :term;
Premises	$\oplus L_2$
Constraint	$H_2 \leftarrow$ listcons(L_1, H_3)
Conclusions	$\ominus L_3, \oplus L_1$
Declarative Content	$L_1 \quad (L_1) \vdash \Phi \quad$ (Hyp) $L_2 \quad H_2 \vdash \Psi \quad (J)$ $L_3 \quad H_3 \vdash \Phi \rightarrow \Psi \quad$ (implies-I L_2)
Procedural Content	schema – interpreter

Method : equiv-I	
Declarations	L_1, L_2, L_3 :prln; H :list(prln); J_1, J_2 :just; Ψ, Φ :term;
Premises	$\oplus L_1, \oplus L_2$
Constraint	t
Conclusions	$\ominus L_3$
Declarative Content	$L_1 \quad H_3 \vdash \Phi \rightarrow \Psi \quad (J_1)$ $L_2 \quad H_3 \vdash \Psi \rightarrow \Phi \quad (J_2)$ $L_3 \quad H_3 \vdash \Phi \leftrightarrow \Psi \quad$ (equiv-I L_2, L_1)
Procedural Content	schema – interpreter

The methods or-I-l and or-I-r suggest to prove an open line with a disjunction $\Phi \vee \Psi$ by proving the left side Φ or the right side Ψ, respectively.

Method : or-I-l	
Declarations	L_1, L_2 :prln; H :list(prln); J :just; Ψ, Φ :term;
Premises	$\oplus L_2$
Constraint	t
Conclusions	$\ominus L_1$
Declarative Content	$L_2 \quad H \vdash \Phi \quad (J)$ $L_1 \quad H \vdash \Phi \vee \Psi \quad$ (or-I-l L_2)
Procedural Content	schema – interpreter

Method : or-I-r	
Declarations	L_1, L_2 :prln; H :list(prln); J :just; Ψ, Φ :term;
Premises	$\oplus L_2$
Constraint	t
Conclusions	$\ominus L_1$
Declarative Content	L_2 $H \vdash \Psi$ (J) L_1 $H \vdash \Phi \vee \Psi$ (or-I-r L_2)
Procedural Content	schema – interpreter

The or-E method is used to eliminate a disjunction in a support line. As in the corresponding ND calculus rule, the actual goal is also needed in this method. The first line in the constraints demands that H_1 is a subset of H_6.

Method : or-E	
Declarations	$L_1, L_2, L_3, L_4, L_5, L_6$:prln; H_1, H_3, H_5, H_6 :list(prln); J_1, J_3, J_5 :just; Ψ, Φ, Ξ :term;
Premises	$\ominus L_1, \oplus L_3, \oplus L_5$
Constraint	$\bigwedge^{H_1} x$:prln.$\bigvee^{H_6} y$:prln.$x = y$ & $H_3 \leftarrow$ listcons(L_2, H_6) & $H_5 \leftarrow$ listcons(L_4, H_6)
Conclusions	$\ominus L_6, \oplus L_2, \oplus L_4$
Declarative Content	L_1 $H_1 \vdash \Phi \vee \Psi$ (J_1) L_2 $(L_2) \vdash \Phi$ (Hyp) L_3 $H_3 \vdash \Xi$ (J_3) L_4 $(L_4) \vdash \Psi$ (Hyp) L_5 $H_5 \vdash \Xi$ (J_5) L_6 $H_6 \vdash \Xi$ (or-E)
Procedural Content	schema – interpreter

The same-method eliminates trivial gaps. If a support line $H_1 \vdash \Phi$ and an open line $H_2 \vdash \Phi$ have been constructed by forward and backward reasoning, the gap can be closed by deleting the open line if $H_1 \subseteq H_2$.

Method : same	
Declarations	L_1, L_2 :prln; H_1, H_2 :list(prln); J :just; Φ :term;
Premises	L_1
Constraint	$\bigwedge^{H_1} x$:prln.$\bigvee^{H_2} y$:prln.$x = y$
Conclusions	$\ominus L_2$
Declarative Content	$L_1 \quad H_1 \vdash \Phi \quad (J)$ $L_2 \quad H_2 \vdash \Phi \quad (\texttt{same}\ L_1)$
Procedural Content	`schema – interpreter`

The next method is an extension of the rule ∀-I that can handle a list of quantified variables at one stroke. Note that this is an implemented extension of the syntax that has not been discussed in chapter 3. Meta-variables that are declared as lists of terms are notated with an overline in this chapter: $\overline{X}, \overline{Y}, \overline{T}$. They can be used in quantifications, in applications (e.g. $\forall \overline{X}.P(\overline{X})$), and with the predefined functions and predicates of the constraint language.

Method : forall-I	
Declarations	L_1, L_2 :prln; H :list(prln); J :just; $\overline{X}$:list(var); Φ, Φ_1 :term;
Premises	$\oplus L_1$
Constraint	$\sigma \leftarrow \mathsf{substaddcomp}(\overline{X}, \mathsf{newconst}(\mathsf{termtype}(\overline{X})), \mathsf{emptysubst})$ $\& \Phi_1 \leftarrow \mathsf{substapply}(\sigma, \Phi)$
Conclusions	$\ominus L_2$
Declarative Content	$L_1 \quad H \vdash \Phi_1 \quad (J)$ $L_2 \quad H \vdash \forall \overline{X}.\Phi \quad (\texttt{forall-I}\ L_1)$
Procedural Content	`schema – interpreter`

The following methods deal with the application of definitions or theorems. These methods instantiate a universal quantified line and apply the resulting equivalence or implication to the specified line. They are variants of the `def-i` and `def-e` method in chapter 4, taking into account additionally that the defined predicate can be placed at the right hand side of the equivalence or that an implication instead of an equivalence is used in the definition.

Method : def-I-1	
Declarations	L_1, L_2, L_3 :prln; H_1, H_3 :list(prln); J_1, J_2 :just; $\overline{X}$:list(var); P :const; Ψ, Ψ_1 :term; $\overline{T}$:list(term);
Premises	$L_1, \oplus L_2$
Constraint	$\bigwedge^{H_1} x$:prln.$\bigvee^{H_3} y$:prln.$x = y$ & $\Psi_1 \leftarrow$ substapply(substaddcomp($\overline{X}, \overline{T}$, emptysubst), Ψ)
Conclusions	$\ominus L_3$
Declarative Content	L_1 $H_1 \vdash \forall \overline{X}.P(\overline{X}) \leftrightarrow \Psi$ (J_1) L_2 $H_3 \vdash \Psi_1$ (J_2) L_3 $H_3 \vdash P(\overline{T})$ (def-I-1 L_1, L_2)
Procedural Content	schema – interpreter

Method : def-back	
Declarations	L_1, L_2, L_3 :prln; H_1, H_3 :list(prln); J_1, J_2 :just; $\overline{X}$:list(var); P :const; Ψ, Ψ_1 :term; $\overline{T}$:list(term);
Premises	$L_1, \oplus L_2$
Constraint	$\bigwedge^{H_1} x$:prln.$\bigvee^{H_3} y$:prln.$x = y$ & $\Psi_1 \leftarrow$ substapply(substaddcomp($\overline{X}, \overline{T}$, emptysubst), Ψ)
Conclusions	$\ominus L_3$
Declarative Content	L_1 $H_1 \vdash \forall \overline{X}.\Psi \rightarrow P(\overline{X})$ (J_1) L_2 $H_3 \vdash \Psi_1$ (J_2) L_3 $H_3 \vdash P(\overline{T})$ (def-back L_1, L_2)
Procedural Content	schema – interpreter

Method : def-E-l'	
Declarations	L_1, L_2, L_3 :prln; H_1, H_2, H_3 :list(prln); J_1, J_2 :just; $\overline{X}, \overline{Y}$:list(var); P :const; Ψ, Ψ_1 :term; $\overline{T}$:list(term); σ :subst;
Premises	$L_1, \ominus L_2$
Constraint	$\sigma \leftarrow \mathsf{unify}(\overline{X}, \overline{T}) \,\&\, \sigma \,\&$ $\Psi_1 \leftarrow \mathsf{substapply}(\sigma, \Psi) \,\&\, H_3 \leftarrow \mathsf{listappend}(H_1, H_2)$
Conclusions	$\oplus L_3$
Declarative Content	$L_1 \quad H_1 \vdash \forall \overline{Y}.P(\overline{X}) \leftrightarrow \Psi \quad (J_1)$ $L_2 \quad H_2 \vdash P(\overline{T}) \quad (J_2)$ $L_3 \quad H_3 \vdash \Psi_1 \quad$ (def-E-l' L_1, L_2)
Procedural Content	schema – interpreter

Method : def-E-r'	
Declarations	L_1, L_2, L_3 :prln; H_1, H_2, H_3 :list(prln); J_1, J_2 :just; $\overline{X}, \overline{Y}$:list(var); P :const; Ψ, Ψ_1 :term; $\overline{T}$:list(term); σ :subst;
Premises	$L_1, \ominus L_2$
Constraint	$\sigma \leftarrow \mathsf{unify}(\overline{X}, \overline{T}) \,\&$ $\Psi_1 \leftarrow \mathsf{substapply}(\sigma, \Psi) \,\&\, H_3 \leftarrow \mathsf{listappend}(H_1, H_2)$
Conclusions	$\oplus L_3$
Declarative Content	$L_1 \quad H_1 \vdash \forall \overline{Y}.\Psi \leftrightarrow P(\overline{X}) \quad (J_1)$ $L_2 \quad H_2 \vdash P(\overline{T}) \quad (J_2)$ $L_3 \quad H_3 \vdash \Psi_1 \quad$ (def-E-r' L_1, L_2)
Procedural Content	schema – interpreter

A.2 Proof Plans

In this section we show an example of an application of the hom1-2 method that results in a hierarchical plan. We want to prove that $\| \cup \perp$, namely the union of the two relations parallel and orthogonal, is symmetric. We use the definitions for $\|$ (line A_2 in figure A.1),

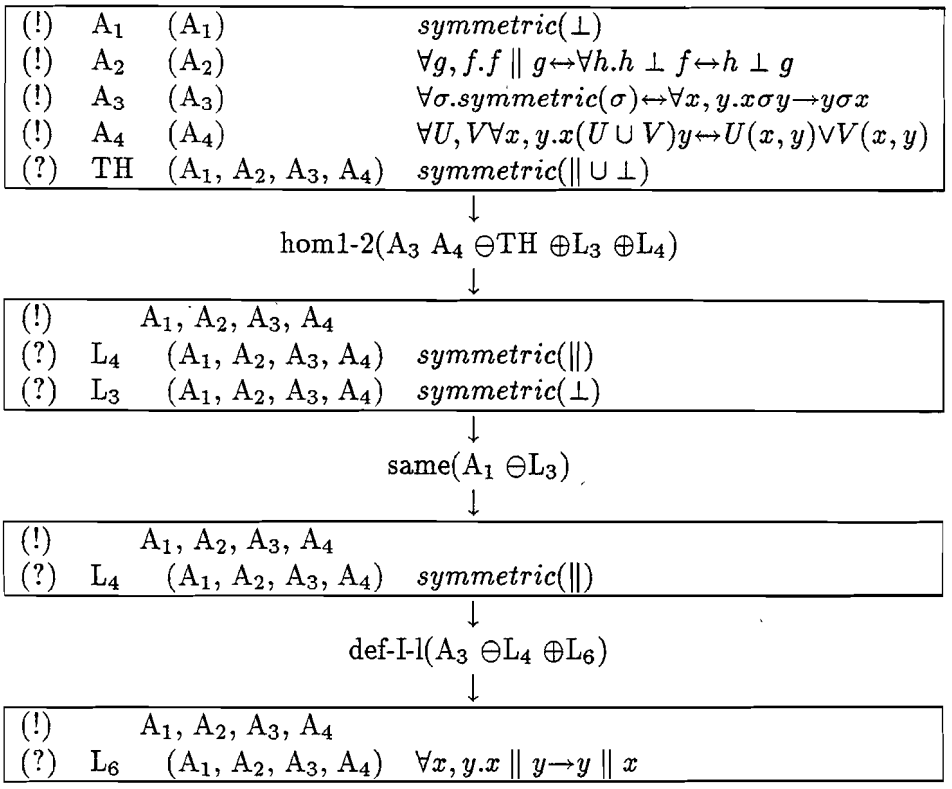


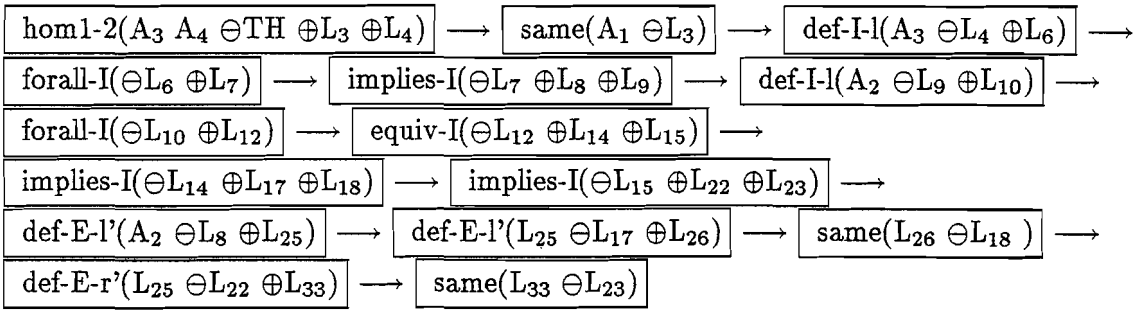
Figure A.1: The planning states and planning operators of the first steps in the proof plan

symmetric (line A_3), and $\cup$ (line A_4). Additionally we use the assumption that $\perp$ is symmetric (line A_1).

The initial planning state and the first planning steps with the resulting planning states can be seen in figure A.1. The initial state consists of the four support lines A_1–A_4 containing the assumptions and the open line TH containing the theorem. In the first step `hom1-2` is applied; it removes TH from the planning state and adds two new open lines, L_3 : $symmetric(\perp)$ and L_4 : $symmetric(\|)$. This step will cause a new call to the planning algorithm after the plan has been completed, because of the the remaining open goal: $symmetric(\| \cup \perp)$ follows from $symmetric(\|)$ and $symmetric(\perp)$. The unlabeled lines A_3 and A_4 in the application of `hom1-1` are not needed during the planning phase but they will be needed when the tactic `hom1-1` is applied.

The first subgoal in L_3 can be solved immediately because it already exists as an assumption; so the method `same` is applied to A_1 and L_3, removing L_3 from the planning state. Then `def-I-l` is applied, expanding the predicate *symmetric* in L_4 with its definition in A_3, constructing a new open line L_6, and deleting L_4. After demonstrating these steps in greater detail, we only give for the rest of the plan the list of planning operators that are applied, and a list of all lines that are used in this plan (figure A.2). The lines are ordered like they would be in the resulting proof. The numbers of the new lines are generated by the planner, that is, they indicate that more lines have been created during the planning process than are used in the final plan. Note that the list of lines is not the final planning state but a list of all lines that are needed in the plan. The final planning

state consists of all support lines except L_8, L_{17}, and L_{22}.



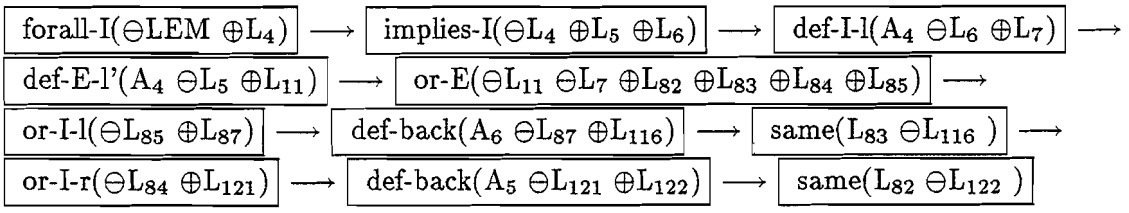
(!)	A_1	(A_1)	$symmetric(\perp)$
(!)	A_2	(A_2)	$\forall g, f.f \parallel g \leftrightarrow \forall h.h \perp f \leftrightarrow h \perp g$
(!)	A_3	(A_3)	$\forall \sigma.symmetric(\sigma) \leftrightarrow \forall x, y.x\sigma y \rightarrow y\sigma x$
(!)	A_4	(A_4)	$\forall U, V \forall x, y.x(U \cup V)y \leftrightarrow U(x,y) \vee V(x,y)$
(!)	L_8	(L_8)	$x_1 \parallel y_2$
(!)	L_{25}	(A_2, L_8)	$\forall h.h \perp x_1 \leftrightarrow h \perp y_2$
(!)	L_{17}	(L_{17})	$h_3 \perp x_1$
(!)	L_{26}	(L_8, A_2, L_{17})	$h_3 \perp y_2$
(!)	L_{22}	(L_{22})	$h_3 \perp y_2$
(!)	L_{33}	(L_8, A_2, L_{22})	$h_3 \perp x_1$
(?)	L_{23}	(L_{22}, L_8, A_1, A_2, A_3, A_4)	$h_3 \perp x_1$
(?)	L_{15}	(L_8, A_1, A_2, A_3, A_4)	$h_3 \perp y_2 \rightarrow h_3 \perp x_1$
(?)	L_{18}	(L_{17}, L_8, A_1, A_2, A_3, A_4)	$h_3 \perp y_2$
(?)	L_{14}	(L_8, A_1, A_2, A_3, A_4)	$h_3 \perp x_1 \rightarrow h_3 \perp y_2$
(?)	L_{12}	(L_8, A_1, A_2, A_3, A_4)	$h_3 \perp y_2 \leftrightarrow h_3 \perp x_1$
(?)	L_{10}	(L_8, A_1, A_2, A_3, A_4)	$\forall h.h \perp y_2 \leftrightarrow h \perp x_1$
(?)	L_9	(L_8, A_1, A_2, A_3, A_4)	$y_2 \parallel x_1$
(?)	L_7	(A_1, A_2, A_3, A_4)	$x_1 \parallel y_2 \rightarrow y_2 \parallel x_1$
(?)	L_6	(A_1, A_2, A_3, A_4)	$\forall x, y.x \parallel y \rightarrow y \parallel x$
(?)	L_4	(A_1, A_2, A_3, A_4)	$symmetric(\parallel)$
(?)	L_3	(A_1, A_2, A_3, A_4)	$symmetric(\perp)$
(?)	TH	(A_1, A_2, A_3, A_4)	$symmetric(\parallel \cup \perp)$

Figure A.2: The proof plan for our example and a list of all used lines

After line L_6 is created, the plan continues by simplifying this line in a straitforward way: the quantifier is removed with `forall-I` and the resulting implication is eliminated with `implies-I`. This results in a planning state with the open line $L_9 : y_2 \| x_1$ and the additional support line $L_8 : x_1 \| y_2$. Now the plan repeats the preceeding procedure with L_9: the predicate $\|$ is expanded with its definition, the quantifier in L_{10} is removed, the equivalence in L_{12} is split, and the implications in L_{14} and L_{15} are split. The resulting planning state contains the open lines $L_{18} : h_3 \perp y_2$ and $L_{23} : h_3 \perp x_1$ and the additional support lines $L_{17} : h_3 \perp x_1$ and $L_{22} : h_3 \perp y_2$. Note that the method `same` can not be applied in this state because the hypotheses of L_{17} are not a subset of the hypotheses of L_{23}, and the same holds for L_{22} and L_{18}.

Up to this point only backward reasoning has been used in the plan, because this is one of our simple heuristics. But now no more methods can be applied to the open lines. So from now on the plan performs forward reasoning: first the predicate $\|$ in L_8 is expanded by the method `def-E-l'`. Then the resulting support line L_{25} is used like a definition: it is applied to L_{17} and L_{22} by `def-E-l'` and `def-E-r'`, respectively. With the resulting lines L_{26} and L_{33} the plan can be completed by two applications of the method `same` which delete the last two open lines L_{18} and L_{23}.

This proof plan for the goal *symmetric*($\| \cup \perp$) uses the `hom1-2` method. After the plan is found and the corresponding tactics are applied in order to construct the proof, the application of the `hom1-2` tactic leaves a gap in the proof. In this gap the expanded definition of *symmetric*($\| \cup \perp$) (see line LEM in figure A.3) has to be proved with the help of the expanded definition of *symmetric*($\|$) and *symmetric*($\perp$) (lines A_5 and A_6). The method `hom1-2` (see line L_7 in figure 5.1) proposes additionally to use the definition of the function $\cup$ (line A_4). This gap is closed with another call to the proof planner. The proof plan and the used lines in this plan are shown in figure A.3.



(!)	A_4	(A_4)	$\forall U, V \forall x, y. x(U \cup V)y \leftrightarrow U(x,y) \vee V(x,y)$
(!)	A_5	(A_5)	$\forall x, y. x \parallel y \rightarrow y \parallel x$
(!)	A_6	(A_6)	$\forall x, y. x \perp y \rightarrow y \perp x$
(!)	L_5	(L_5)	$x_1(\perp \cup \|)y_2$
(!)	L_{11}	(A_4, L_5)	$x_1 \perp y_2 \vee x_1 \parallel y_2$
(!)	L_{82}	(L_{82})	$x_1 \parallel y_2$
(!)	L_{83}	(L_{83})	$x_1 \perp y_2$
(?)	L_{116}	(L_{83}, L_5, A_4, A_5, A_6)	$x_1 \perp y_2$
(?)	L_{87}	(L_{83}, L_5, A_4, A_5, A_6)	$y_2 \perp x_1$
(?)	L_{85}	(L_{83}, L_5, A_4, A_5, A_6)	$y_2 \perp x_1 \vee y_2 \parallel x_1$
(?)	L_{122}	(L_{82}, L_5, A_4, A_5, A_6)	$x_1 \parallel y_2$
(?)	L_{121}	(L_{82}, L_5, A_4, A_5, A_6)	$y_2 \parallel x_1$
(?)	L_{84}	(L_{82}, L_5, A_4, A_5, A_6)	$y_2 \perp x_1 \vee y_2 \parallel x_1$
(?)	L_7	(L_5, A_4, A_5, A_6)	$y_2 \perp x_1 \vee y_2 \parallel x_1$
(?)	L_6	(L_5, A_4, A_5, A_6)	$y_2(\perp \cup \|)x_1$
(?)	L_4	(A_4, A_5, A_6)	$x_1(\perp \cup \|)y_2 \rightarrow y_2(\perp \cup \|)x_1$
(?)	LEM	(A_4, A_5, A_6)	$\forall x, y. x(\perp \cup \|)y \rightarrow y(\perp \cup \|)x$

Figure A.3: The proof plan for the gap left open by `hom1-2` in the first proof plan

In the first two steps the quantifier in LEM is removed and the implication is eliminated. Then the definition of $\cup$ is applied to L_6 (backward reasoning by `def-I-l`) and to L_5 (forward reasoning by `def-E-l'`). After this `or-E` is applied to L_7 and L_{11}, splitting

the supporting disjunction into L_{82} and L_{83}, deleting the goal L_7, and creating two new subgoals L_{84} and L_{85}, each having one of the disjuncts L_{82} and L_{83} as additional hypotheses. Both subgoals are proved analogously: the left disjunct of L_{85} is proved with `or-I-l` (the right disjunct of L_{84} with `or-I-r`,respectively), in order to prove the disjunction; then the assumption A_6 (respectively A_5) is applied with `def-back`; and finally the gap is closed with `same`.

After executing the two plans consecutively by carrying out the tactic part of the methods, all gaps are closed and the result is a complete proof for *symmetric*($\| \cup \perp$).

Bibliography

[1] Peter B. Andrews. Transforming matings into natural deduction proofs. In Wolfgang Bibel and Robert Kowalski, editors, *Proceedings of the 5th CADE*, Les Arcs, France, 1980. Springer Verlag, Berlin, Germany. LNCS 87.

[2] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Academic Press, Orlando, Florida, USA, 1986.

[3] Robert S. Boyer and J Strother Moore. *A Computational Logic.* Academic Press, New York, USA, 1979.

[4] Nicolaas Govert de Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry - Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, London, United Kingdom, 1980.

[5] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, **7**:303–324, 1991. Also pubished as DAI Research Paper No. 413.

[6] Alan Bundy. The use of explicit plans to guide inductive proofs. In *Proceedings of the 9th CADE*, Argonne, Illinois, USA, 1988. Springer Verlag, Berlin, Germany. LNCS 310.

[7] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The OYSTER-CLAM system. In Mark E. Stickel, editor, *Proceedings of the 10th CADE*, pages 647–648, Kaiserslautern, Germany, 1990. Springer Verlag, Berlin, Germany. LNAI 449.

[8] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, **5**:56–68, 1940.

[9] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1986.

[10] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. In Mark E. Stickel, editor, *Proceedings of the 10th CADE*, pages 653–654, Kaiserslautern, Germany, 1990. Springer Verlag, Berlin, Germany. LNAI 449.

[11] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, **2**:189–208, 1971.

[12] Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, **39**:176–210, 1935.

[13] Gerhard Gentzen. Untersuchungen über das logische Schließen II. *Mathematische Zeitschrift*, **39**:572–595, 1935.

[14] Fausto Giunchiglia and Paolo Traverso. Program tactics and logic tactics. IRST-Technical Report 9301-01, Istituto per la Ricerca Scientifica e Tecnologica, Trento, Italy, 1993. also in Proceedings of LPAR-94, *5th International Conference on Logic Programming and Automated Reasoning*, Kiev, Ukraine, July 16-21, 1994.

[15] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation.* LNCS 78. Springer Verlag, Berlin, Germany, 1979.

[16] Xiaorong Huang. Reconstructing proofs at the assertion level. In Alan Bundy, editor, *Proceedings of the 12th CADE*, pages 738–752, Nancy, France, 1994. Springer Verlag, Berlin, Germany. LNAI 814.

[17] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jörn Richts, and Jörg Siekmann. Ω-MKRP: A proof development environment. In Alan Bundy, editor, *Proceedings of the 12th CADE*, pages 788–792, Nancy, 1994. Springer Verlag, Berlin, Germany. LNAI 814.

[18] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, and Jörn Richts. Adapting methods to novel tasks in proof planning. In Bernhard Nebel and Leonie Dreschler-Fischer, editors, *KI-94: Advances in Artificial Intelligence – Proceedings of KI-94, 18th German Annual Conference on Artificial Intelligence*, pages 379–390, Saarbrücken, Germany, 1994. Springer Verlag, Berlin, Germany. LNAI 861.

[19] Michael Kohlhase. Unification in order-sorted type theory. In A. Voronkov, editor, *Proceedings of LPAR*, pages 421–432, Berlin, Germany, 1992. Springer Verlag. LNAI 624.

[20] Dan Nesmith, editor. KEIM-Manual. Version 1.2, 1994. Universität des Saarlandes, Im Stadtwald, Saarbrücken, Germany.

[21] Allen Newell. The heuristic of George Polya and its relation to artificial intelligence. Technical Report CMU-CS-81-133, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, USA, 1981. also in Rudolf Groner, Marina Groner and Walter F. Bishoof, Editors, Methods of Heuristics, Lawrence Erlbaum, Hillsdale, New Jersey, USA, pages 195–243.

[22] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. *Logic and Computer Science*, pages 361–386, 1990.

[23] George Pólya. *How to Solve It.* Princeton University Press, Princeton, New Jersey, USA, also as Penguin Book, 1990, London, United Kingdom, 1945.

[24] George Pólya. *Mathematics and Plausible Reasoning.* Princeton University Press, Princeton, New Jersey, USA, 1954. Two volumes, Vol.1: Induction and Analogy in Mathematics, Vol.2: Patterns of Plausible Inference.

[25] Dag Prawitz. *Natural Deduction - A Proof Theoretical Study.* Almqvist & Wiksell, Stockholm, Sweden, 1965.

[26] V. S. Subrahmanian. A Simple Formulation of the Theory of Metalogic Programming. In Harvey Abrahmason and M. H. Rogers, editors, *Meta–Programming in Logic Programming*, chapter 4, pages 65 – 101. The MIT Press, 1989.

[27] Kurt VanLehn. Problem solving and cognitive skill acquisition. In Michael I. Posner, editor, *Foundations of Cognitive Science*, chapter 14. MIT Press, Cambridge, Massachusetts, 1989.