Interactive Theorem Proving with Indexed Formulas

Malte Hübner

Diplomarbeit

Saarbrücken, April 2003

FR Informatik
Universität des Saarlandes
Postfach 15 11 50
66041 Saarbrücken

Acknowledgements

I am indepted to Christoph Benzmüller for his dedicated supervision in the last years and in particular during this project. Serge Autexier has patiently answered any questions about CORE. Thanks go also to Andreas Meier and the other members of the ΩMEGA-group for the many helpful discussions. I am also grateful to my parents for their support throughout my studies.

Erklärung

Hiermit erkläre ich an Eides statt, dass ich diese Arbeit selbstständig angefertigt und keine anderen als die angegebenen Quellen verwendet habe.

Saarbrücken, den 05. April 2003

Malte Hübner

Contents

English Abstract

Since more than two decades research in interactive theorem proving (ITP) has attracted growing interest. The primary application domains for ITPs range from hard– and software verification tools to mathematical tutor systems. To support communication with the user in an adequate way these systems depend on calculi that allow for the construction of human understandable and readable proofs. However, most calculi that are used in current ITPs fall still short of supporting the user in an optimal way. The reason is that they enforce the user to construct proofs at a level that is far more detailed than the one that can be found in human constructed proofs.

Autexier [Aut03] has recently proposed a new theorem proving framework that allows to model different logics and calculi in an uniform way. In CORE, a proof-state is always represented as a single formula that can be manipulated by the application of replacement rules that are generated from the logical context of the subformula under transformation. This approach also facilitates proof construction at the assertion level which is considered as more closely matching the level at which humans construct proofs (see for instance [Hua94]). Together with COREs window inference technique this makes CORE a potentially well suited basis for interactive theorem proving.

This thesis tries to excerpt COREs potential for interactive theorem proving by mapping important concepts of the established proof system ΩMEGA to CORE. A task structure is developed to present the context of a subformula in an intuitive way to the user and to assist him in structuring proofs. The development of a method interpreter makes it possible to specify abstract inference steps declaratively and to encode proof strategies for the use in CORE. The adaptation of ΩMEGAs agent-based suggestion mechanism ΩANTS to CORE helps the user with the identification of applicable methods and replacement rules.

Deutsche Kurzzusammenfassung

Interaktives Theorembeweisen hat in den letzten zwei Jahrzehnten zunehmend an Bedeutung gewonnen. Die Anwendungsbereiche von Systemen mit denen sich Beweise interaktiv führen lassen reichen von der Hard- und Software Verifikation bis zu mathematischen Tutor-Systemen. Die genannten Anwendungsgebiete machen es erforderlich das der Anwender bei der Beweisführung adequat untersützt wird. Um eine entsprechende Kommunikation mit dem Benutzer zu ermöglichen verwenden interaktive Beweissysteme Kalküle, in denen Beweise in einer für den Benutzer nachvollziehbaren Art und Weise, geführt werden können. Trotzdem kann man noch nicht davon sprechen, dass interaktive Beweiser den Benutzer optimal unterstützen. Der Hauptrgund hierfür ist, dass die eingesetzten Kalküle automatisch dazu führen, dass Beweise auf einer viel detailliertern Ebene geführt werden müssen, als man typischerweise in einem mathematischen Beweis finden würde.

Autexier [Aut03] hat kürzlich eine neue logische Umgebung für die Beweissuche entwickelt, welche es ermöglicht verschiedene Logiken und Kalküle einheitlich zu modellieren. In CORE ist ein Beweiszustand immer als eine einzige Formel repräsentiert, welche durch das Anwenden von Ersetzungsregeln, die aus dem Kontext einer Teilformel abgeleitet werden, transformiert werden kann. Diese Herangehensweise erleichtert es auch, Beweise auf der sogenannten Assertion-Ebene (vgl. [Hua94]) zu führen, welche allgemein als eine natürlichere Ebene für die Beweisführung angesehen wird. Zusammen mit der Window-Inferenz Technik, die von CORE unterstützt, wird stellt CORE ein System dar das potentiell als eine verbesserte Grundlage für die interaktive Beweissuche angesehen werden kann.

In dieser Arbeit geht es darum, das Potential von CORE im Hinblick auf die interaktive Beweiskonstruktion auszunutzen. Dieses geschieht zum einen dadurch, dass etablierte Konzepte im Bereich des interaktiven Beweisens, wie sie auch im Beweis-System ΩMEGA verwendet werden, auf das CORE-System abgebildet werden. Desweiteren wird eine Task-Srukture entwickelt, die es zum einen ermöglicht, den logischen Kontext einer Teilformel für den Benutzer verständlich aufzubereiten und darzustellen; zum anderen untersützt sie den Anwender auch darin, Beweise strukturiert zu führen. Der Entwurf eines Methoden-Interpreters

ermöglicht es, abstrakte Beweisschritte zu kodieren und im System anzuwenden. Die Anpassung des Vorschlags-Mechanismus ΩANTS an CORE stellt eine weitere Unterstützung für den Benutzer bereit, indem sie automatisch Vorschläge über mögliche Fortsetzungen eines Beweises generiert.

Chapter 1

Introduction

From the very beginning, automated deduction (or automated theorem proving) was a key research area within Artificial Intelligence (see for instance [GHL60]). Although the automatic generation of proofs for mathematical conjectures was a goal in its own right, automated deduction drew additional importance from the fact that it was considered as a means to automate commonsense reasoning.

While early research on automated deduction was solely interested in fully automated search for derivations, one can now broadly classify research on automated deduction along two categories: machine-oriented proof search and human-oriented proof search.

Machine-oriented proof search continues with the tradition of inventing calculi and search strategies that enable computers to search efficiently for proofs without requiring any user-interaction. Research into this direction has lead to the development of machine-oriented calculi such as resolution [Rob65], tableaux [Smu68] and matrix [Bib82] proof search, of which resolution and its derivants are probably the most successful.

Although automated provers that make use of these machine-oriented calculi had some success in proving previously unproved theorems like *Robbins Algebra Conjecture* [McC97] they are subject to at least two lines of criticism. Firstly, despite being able to successfully prove some theorems, the task of searching for proofs often turned out to be too complex to prove even simple theorems. Secondly, because the calculi underlying automated provers use a machine-oriented representation of the problem, proofs that are returned by these systems are hardly readable for humans.

These limits of automated theorem provers have sparked interest in so called *interactive theorem proving*. Here, the task of finding a proof is divided between user and computer. Typically the user has to construct a proof by applying inference rules of a more user friendly calculus while the computer guarantees the correctness of the so constructed proof. Some systems go even further and support the user by checking which rules are applicable in a given proof state or even make suggestions about which rule to apply next [BS99a].

To facilitate communication with the user, interactive theorem provers (ITPs) make use of calculi that are thought to better match the human style of reasoning than do calculi that are developed for fully automated proof search. This enhances the readability of the proofs produced and facilitates presentation of a proof or proof state to the user. Currently, the most frequently used calculus is Gentzens *natural deduction calculus (ND)* or its relative the *sequent calculus* [Gen35], or variants of it. For instance, systems like the Carnegie Proof Tutor, ISABELLE [Pau86] or ΩMEGA [BCF$^+$97] all use a ND variant as underlying calculus. Other system, like COQ [BBC$^+$97] are internally based on constructive type theory, but use an ND calculus for the communication with the user.

However, even with more user-friendly calculi and suggestion mechanisms, such as ΩANTS [BS99a], conducting proofs in an interactive system is still a challenging task. One reason for this is that although the calculi employed in ITPs aim at supporting a human-oriented style of reasoning, they still force the user to derive a proof at a much more fine-grained layer then the one that is usually used by human mathematicians. As a result, interactively constructed proofs require to perform many calculus intrinsic proof steps explicitely that occur implicitly in human created proofs. An example is the explicit decomposition of assertions [Hua94], which is necessary in the sequent and natural deduction calculus. Humans would immediately focus on the relevant subtask, without having to extract it by employing lengthy but trivial inference steps.

These problems have led to attempts to structure the proof search at a more abstract level which resulted in the development of *tactical theorem proving* [Mil85] and *proof planning* [Bun88]. In tactical theorem proving, proof construction mainly proceeds by the application of tactics, little programs that transform the proof state by automatically applying a sequence of inference rules and allow for the encoding of frequently recurring proof patterns or even human-oriented proof techniques, such as the induction principle, etc. In proof-planning this procedural encoding of domain-dependent knowledge is augmented by a declarative specification which makes it possible to use the abstract proof steps as planning operators (*proof methods*). However, non of the new paradigms have turned out to be the ultimate solution (see [Bun02] for a critic on proof-planning, also [BMM$^+$01]).

Autexier [Aut01, Aut03] recently developed the CORE theorem proving system which was built as a basis for the integration of multiple proof paradigms into one framework. The system supports a *context* oriented proof style that is related to ideas of the *window inference technique* of [RS93]. The novelty of the system lies in the fact that it does not make use of a fixed set of calculus rules that are defined over the syntactic structure of formulas; rather, it allows the user to focus the reasoning process on subformulas of the overall goal formula. Information contained in the context of the focus is then made available to the user in form of transformation rules, so called *replacement rules*, that can be used to transform the subformula in the current focus without enforcing a decomposition of the overall formula. One of the advantages of this reasoning style is that it supports the application of assertions in a more intuitive way. In fact, the stepwise unwrapping of subgoals is replaced by relocation of the focus and application of replacement rules.

Accordingly, proof search in the new system proceeds by i) placing the focus onto subformulas of the goal formula and ii) applying replacement rules to subformulas in the focus. Replacement rules are of the form

$$i \to< v_1, \ldots, v_n >$$

where $i, v_1, \ldots, v_n$ are logical formulas. Rules of this form replace a formula i, called the *input* of the rule, by a conjunction of formulas $v_1, \ldots, v_n$, the *values* of the rule.

However, although the system in principle allows for conducting proofs in a more 'natural' way than other ITPs, there is still additional work required to make full use of COREs potential for interactive reasoning. Four issues are particularly evident.

1. In every proof state there is a great number of rules available for transforming the focused formula, each corresponding to a particular application direction of an assertion in the context of the focus. However, in general, only a fraction of these rules is *applicable* in the respective proof state. It is therefore a problem to identify the applicable replacement rules amongst all available rules.

2. This also means that when the user wants to apply a particular definition or lemma (assertion) to the subproblem in focus he has no direct access to the assertions in the context of that subproblem. In order to apply an assertion he has to identify the particular replacement rule amongst the probably many applicable ones that corresponds to the intended application direction of the assertion. Ideally, he could first select an assertion for application and then concentrate on finding the rule for the required application direction of the assertion.

3. Focusing on the appropriate subformula often eases the process of proving a theorem in CORE. Currently, the system provides no guidance as to where to place the focus of the proof. In particular, the user is not prevented from switching foci arbitrarily instead of in a systematic way.

4. At the moment COREs facilities for performing abstract proof steps are insufficient; i.e. abstract proof steps can only be represented in the form of programs (so called tactics), but not in a declarative way (as is for instance the case in the ΩMEGA system).

The contribution of this thesis can be divided into three related issues. Firstly, we develop a datastructure of so called *tasks* that make it possible to display a subgoal of a proof together with all assertions available for proving this subgoal to the user. Tasks provide explicit access to the assertions in the context and will therefore allow us to first select an assertion for application and then generate just those replacement rules that realize the application of the assertion. We will see that in many cases it is straight forward to determine which replacement rule is appropriate for application of the chosen assertion.

This means that instead of having to select one out of all admissible rules for application we only have to choose from those replacement rules that realize the application of the assertion. Furthermore, we will develop *tasks* in a way that they resemble the tasks used in ΩMEGAs proof planner MULTI [Mei03]. This will make it possible to switch between proof planning and interactive proof construction within a single proof attempt without having to change the representation layer.

Secondly, we will be concerned with developing a notion of a method for the CORE system. Methods make it possible to encode specific proof-techniques or abbreviations of frequently occurring subproofs in a declarative language and apply them as a single proof step in CORE. In fact, the methods that we will introduce in this thesis are a uniform concept which comprises the concepts of methods and tactics that are used in ΩMEGA and can hence also be used as planning operators by a proof-planner as well as for interactive proof construction. When defining methods we will already make use of the task structure introduced earlier.

Thirdly, we realize an adaptation of ΩMEGAs agent-based suggestion mechanism ΩANTS [BS99a] to the new system. In ΩMEGA this mechanism checks which inference rules (i.e. calculus rules and tactics) are applicable in a given proof state and suggests the most promising rules to the user. ΩANTS performs its computations in parallel between two user interactions and is in principle calculus independent. We will adapt this mechanism to support interactive proof construction in CORE. A main problem in this respect is to deal with the dynamically generated inference rules.

Although CORE is a higher-order framework it is treated as a first-order system in this thesis to make it easier to present the fundamental aspects. However, all concepts developed in this thesis scale to the higher-order case.

Accordingly, the thesis is laid out as follows. Chapter 2 and Chapter 3 prepare the ground on which subsequent chapters are based. Chapter 2 introduces the basic terminology and in Chapter 3 we introduce the basic concepts of the CORE framework. In Chapter 4 we develop the task-datastructure. This datastructure will serve as a new communication layer on top of the CORE system which helps us to structure proofs. The then following Chapter 5 steps through a sample proof to demonstrate the advantages that this datastructure brings for interactive proof search. In Chapter 6 we will be concerned with defining the notion of a proof method for the CORE system. The methods we introduce in this chapter are declarative specifications of abstract proof steps. In chapters 7 and 8 we will see how we can use this declarative specification to automatically test the applicability of methods in a given proof state. To perform these tests and to make suggestions about which method to apply we will adapt ΩMEGAs suggestion mechanism ΩANTS to CORE. Chapter 8 describes how this is done, while Chapter 7 introduces the ΩANTS mechanism.

Chapter 2

Preliminaries

Although we assume basic familiarity with first-order logic (see for instance [Fit96]), sequent- and natural deduction calculus [Gen35] as well as tactical theorem proving this chapter briefly introduces the basic concepts that are used in this thesis. We will begin with fixing a language $\mathcal{L}$ of first-order logic formulas before introducing the terminology related to the sequent calculus. This will later turn out to be very useful for the understanding of the CORE framework which is best explained by comparison to the sequent calculus.

2.1 Syntax of First-Order Logic

When speaking about logical formulas we always assume that they are elements of the language $\mathcal{L}$ defined below. The basic constructs out of which $\mathcal{L}$ is constructed are the following

a) Countably infinitely many variables.

b) Countably infinitely many constants.

c) Countably many n-ary predicate symbols $P^n, Q^n, R^n, \ldots$

d) Countably many k-ary function symbols $f^k, g^k, h^k, \ldots$

In general we denote constants as $a, b, b, \ldots$ and variables as $x, y, z, \ldots$ (or $M, N, \ldots$ when we are dealing with sets).

We define the *terms* of our language as all symbols denoting variables and constants as well as all $k+1$-tuples $(f^k, t_1, \ldots, t_k)$ where the t_i are again terms. *Atomic formulas* are all $n+1$-tuples $(P^n, t_1, \ldots, t_n)$, where t_i are terms and P^n is an n-ary predicate symbol. We display atomic formulas as $P(t_1, \ldots, t_n)$ and omit the superscript n whenever the arity is clear from the context. Similarly we display terms $(f^k, t_1, \ldots, t_k)$ as $f(t_1, \ldots, t_k)$.

We can now define the language $\mathcal{L}$ as follows.

Definition 2.1.1 *The set $\mathcal{L}$ is the smallest set for which the following holds:*

1. *All atomic formulas are in $\mathcal{L}$.*
2. *If A is in $\mathcal{L}$, so is $\neg A$.*
3. *If A and B are in $\mathcal{L}$ then $A \wedge B$, $A \vee B$ and $A \Rightarrow B$ are in $\mathcal{L}$.*
4. *If x is a variable and $A \in \mathcal{L}$ then $\exists x.A$ and $\forall x.A$ are in $\mathcal{L}$.*

In order to avoid bracketing we give $\wedge$ precedence over $\vee$ and $\neg$ precedence over $\wedge$. $\Rightarrow$ has lowest precedence.

Furthermore we will consider as a *literal* all atomic formulas A and their negation $\neg A$. We define the set of *subformulas* of a formula.

Definition 2.1.2 *Let F be a first-order formula from the language $\mathcal{L}$. Then the set $SF(F)$ of all* subformulas *of F is the smallest set such that:*

1. *F is in $SF(F)$.*
2. *If $\neg G$ is in $SF(F)$, then $G \in SF(F)$.*
3. *If $G_1 \wedge G_2$, $G_1 \vee G_2$ or $G_1 \Rightarrow G_2$ is in $SF(F)$ then $G_1, G_2 \in SF(F)$.*
4. *If $\forall x.G$ or $\exists x.G$ is in $SF(F)$ then $G \in SF(F)$.*

If a formula F is of the form $\neg G$ or $Qx.G$ for $Q \in \{\exists, \forall\}$ then we call G the *major subformula* of F. Similarly F and G are the major subformulas of $F \wedge G$, $F \vee G$ and $F \Rightarrow G$.

We will further assume familiarity with the concept of substitution and write $F[c/x]$ for denoting that all free occurrences of x are replaced by c in F.

2.2 Term Positions

In the subsequent chapters we occasionally need to make reference to subformulas inside a formula. To see how this can be achieved we introduce *term positions* in two steps. First, we define what a *valid* term position is and then we describe which subterm of a formula is referenced by a given term position.

Definition 2.2.1 *(Valid Term Position) A* term position *is a possibly empty list of natural numbers $[i_0, \ldots, i_n]$.*

Valid term positions for terms t are the following.

- *If t is a variable or constant then $\pi = []$ is the only valid term position for t.*
- *If $t = f(t_1, \ldots, t_n)$ and π_i is a valid term position for t_i, $i = 1, \ldots, n$ then $[i, \pi_i]$ is a valid term position for t.*

For formulas $F \in \mathcal{L}$ the valid term positions are defined as follows.

- *If F is an atomic formula, i.e. $F = P(t_1, \ldots, t_k)$ and π_i is a valid term position for t_i, $i = 1, \ldots, n$ then $[i, \pi_i]$ is a valid term position for F.*

- *If $F = \neg G$ and π is a valid term position for G then $[1, \pi]$ is a valid term position for F.*

- *If $F = G_1 \circ G_2$, $\circ \in \{\wedge, \vee, \Rightarrow\}$ and π_1, π_2 are valid term positions for G_1 and G_2 respectively then $[1, \pi_1]$ and $[2, \pi_2]$ are valid term positions for F.*

- *If $F = Qx.G[x]$ for $Q \in \{\forall, \exists\}$ and π is a valid term position for $G[x]$ then $[1, \pi]$ is a valid term position for F.*

If π is a valid term position for F we denote by $F_{|\pi}$ the subformula (or subterm) referenced by π, i.e.

- $F_{[]} = F$

- $P(t_1, \ldots, t_k)_{|[i]} = t_i$ and $f(t_1, \ldots, t_k)_{|[i]} = t_i$ for $i = 1, \ldots, k$, a predicate P and a function f

- $(G_1 \circ G_2)_{|[i,\pi]} = G_{i|\pi}$ for $\circ \in \{\wedge, \vee, \Rightarrow\}$ and $i = 1, 2$.

- $Qx.G[x]_{|[1,\pi]} = G[x]_{|\pi}$ if $Q \in \{\forall, \exists\}$.

For a formula like $F = A \vee (B \Rightarrow Q(g(a), b))$ we now have $F_{|[2,2,1,1]} = a$ and $F_{|[1]} = A$.

Using the above definition we introduce $F[t]_\pi$ as a notation for $F_{|\pi} = t$, i.e. t occurs as subterm in F at position π. $F[t]$ simply says that there exists a position π such that $F_{|\pi} = t$. We extend this notation to subformulas S; i.e. $F[S]_\pi$ means that $F_{|\pi} = S$.

2.3 Sequent Calculus

The *sequent* calculus (SK) was initially defined by Gentzen [Gen35] [1] . However, in this thesis we follow the definition by Wallen [Wal90]. A *sequent* is thus defined as an ordered pair (Σ, Δ) of sets of formulas. We use the common notation and represent a sequent as $\Sigma \vdash \Delta$. Furthermore, Σ, A and Δ, A denote the sets $\Sigma \cup \{A\}$ and $\Delta \cup \{A\}$ respectively. We refer to sequents of the form $\Sigma, A \vdash A, \Delta$ as *initial sequents.*

Formulas $F \in \Sigma \cup \Delta$ are referred to as *sequent formulas* (s-formula). Note that a subformula of an s-formula is not an s-formula of the same sequent.

Sequents are interpreted in the traditional way namely that a model $\mathcal{M}$ satisfies a sequent $\Sigma \vdash \rhd$ if it holds that when $\mathcal{M}$ satisfies all formulas in Σ then it also satisfies at least one formula in Δ.

[1] Originally, Gentzen devised the sequent calculus only as a technical aid to deal with the problem of Cut-Elimination in Natural Deduction.

2.4 Natural Deduction

In this thesis we do not require a detailed knowledge about the *natural deduction* calculus (ND). However, for an introduction to ND we refer to [Fit96]. We will often make reference to the natural deduction variant of the ΩMEGA system. In this case we always have in mind the calculus as defined in [Sor01]. When we have to display proof segments of the ND calculus we do this in linearized notation as introduced by [And80]. A proof fragment in the linearized ND calculus is a finite set of proof lines, where each proof line is of the form

$$L.\Delta \vdash F \ (R)$$

In this notation L is a unique label of the proof line, $\Delta \vdash F$ is a sequent denoting the label F of the proof line along with a set of local hypothesis Δ which can be used to derive F. R is the justification of the line and describes how the line was derived in a proof; i.e. in ΩMEGA proof lines can be justified by ND rules, tactics and methods.

Chapter 3

Introducing CORE

Because in this thesis we are concerned with the facilitation of interactive proof construction in CORE, this chapter gives an overview over the system. The system itself is rather complex and is exhaustively described in [Aut03]. However, to keep this thesis self contained we introduce COREs key characteristics and those aspects that are relevant for this thesis.

The main motivation for the development of CORE was to provide a uniform framework in which various proof construction paradigms can be integrated and compared. For instance, CORE already supports procedural (tactical) theorem proving and we will show in Chapter 6 that it is possible to integrate the concept of declarative proof constructors (methods) to the CORE system. Autexier [Aut03] shows that it is not only possible to support different proof search paradigms in CORE, but also to emulate different calculi (i.e. natural deduction and SK). However, we will ignore this feature in the remainder of this thesis. What is of more importance to us are the following aspects of the CORE system that make it well suited for interactive theorem proving.

1. CORE facilitates direct reasoning at the assertion level (see [Hua94]) which corresponds to the more abstract level at which humans often justify and communicate their proofs, as opposed to the very fine grained layer of calculus rule application.

2. CORE makes it possible to transform subformulas by exploiting the "knowledge" that is contained in their *context*. This knowledge is made available as transformation rules of the form $i \rightarrow < v_1, \ldots, v_n >$. These so called *replacement rules* are generated from the context of a formula and can be used to replace an expression i by the expressions $v_1, \ldots, v_n$, such that the overall formula is transformed into a refined formula. This style of reasoning does not require to decompose the goal formula or to transform it into normal form as is necessary in traditional calculi (e.g. resolution and sequent calculus). As a consequence, a proof state can always be represented as a single formula which makes it possible to present a proof state in a convenient manner to the user.

To realize the proof style described above, CORE annotates formulas and subformulas with proof theoretic information, such as *uniform types* and *polarities.* Based on this annotations one can define the concept of a *logical context* of a formula which is central to the understanding of the CORE system.

In sections 3.2 and 3.3 we will introduce the important notions of *logical contexts* and *replacement rules* before we describe the CORE calculus rules in Section 3.5.

3.1 Proof Theoretic Annotations

We begin with a description of how formulas and subformulas are annotated in CORE which leads us to the notion of *signed formulas* and *indexed formula trees* (IFTS)

3.1.1 Indexed Formula Trees

The concept of an indexed formula tree (which goes back to Wallen [Wal90]) is central to the understanding of the CORE system. Using indexed formula trees in CORE makes it possible to support the contextual reasoning style CORE is aiming at. Furthermore, they provide a means to maintain the goal formula in its entirety, without having to decompose it into sequents or lists of open goals.

Indexed formula trees as defined in [Wal90] are based on the notion of signed formulas and their annotation with uniform types of Smullyan's *uniform notation* [Smu68].

Definition 3.1.1 *(Signed Formulas) A* signed formula *is a pair $< A, p >$, where A is a formula in $\mathcal{L}$ and $p \in \{-1, 1\}$ the* polarity *of A. We often write A^p for $< A, p >$ and sometimes display -1 as $-$ and 1 as $+$.*

Note that if the polarity of a formula is undefined, as is for instance the case for the constituents of a negative equation, we indicate the undefined polarity by 0 (e.g. $(A^0 \Leftrightarrow B^0)^-$).

Signed formulas are assigned a uniform type to distinguish between conjunctive formulas (type α), disjunctive formulas (type β), universal formulas (type γ) and existential formulas (type δ). Table 3.1-3.3 define the type of each signed formula as well as how types are inherited to major subformulas of a signed formula. For the following we agree to denote by $\alpha(\alpha_1^{p_1}, \alpha_2^{p_2})^p$ a signed formula F of type α and polarity p. $\alpha_i^{p_i}$ are the major subformulas of F with polarities p_i. Formulas of type β, γ and δ are denoted in a similar way. To indicate the type of a formula we frequently attach the type information to the leading connective of the formula, e.g. $(A \wedge^\beta B)^+$. We abbreviate $\alpha(F_1, \alpha(F_2, \ldots, \alpha(F_{n-1}, F_n)))$ as $\alpha(F_1, \ldots, F_n)$ and use a respective notation for β. The annotation of signed formulas with types intuitively describes the "behavior" of that formula in a sequent calculus derivation. The polarity $(+/-)$ of a signed formula is just another representation of the succedent/antecedent distinction made by Gentzen [Gen35] with respect to the sequent calculus. The

α	α_1	α_2
$(A \wedge B)^-$	A^-	B^-
$(A \vee B)^+$	A^+	B^+
$(A \Rightarrow B)^+$	A^-	B^+
$(\neg A)^+$	A^-	
$(\neg A)^-$	A^+	

β	β_1	β_2
$(A \wedge B)^+$	A^+	B^+
$(A \vee B)^-$	A^-	B^-
$(A \Rightarrow B)^-$	A^+	B^-

Table 3.1: Uniform Notation (Propositional Types)

interpretation which Wallen [Wal90] has in mind is simply the following: instead of distinguishing between formulas in the antecedent Γ and succedent Δ by using the sequent symbol $\vdash$ (i.e. $\Gamma \vdash \Delta$), he simply annotates all formulas that would occur in the antecedent Γ of a sequent (after application of the appropriate sequent calculus rules) with a negative polarity $(-)$, while formulas that would occur in the succedent are of positive $(+)$ polarity. The uniform type of a formula describes whether the subformulas into which the formula would be decomposed after application of the corresponding sequent calculus rule will occur together in the same sequent (i.e. no split in proof) or will be parts of different sequents (i.e. the proof branches). In the former case the formula is assigned the uniform type α while in the latter case it has uniform type β.

As an example consider the formula $(A \wedge B)$. Depending on the polarity of the formula, that is on which side of a sequent it occurs, this formula has to be decomposed through application of one of the SK rules

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge_L \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \wedge_R \tag{3.1}$$

When $A \wedge B$ occurs on the right hand side of the sequent it has to be decomposed with the rule $\wedge_R$ (3.1, right) which results in a split of the sequent $\Gamma \vdash A \wedge B, \Delta$. Hence, in case $A \wedge B$ has positive polarity it gets assigned type β. However, if the conjunction occurs on the left-hand side of a sequent it has to be decomposed with the rule $\wedge_L$. In this case the corresponding sequent calculus derivation does not branch. Accordingly $A \wedge B$ is of type α in this case.

Furthermore, the types δ and γ are assigned to formulas with a quantifier as leading symbol (i.e. formulas of the form $Qx.F[x]$ with $Q \in \{\forall, \exists\}$). In the SK these formulas have to be decomposed by application of quantifier elimination rules. Here one generally distinguishes between rules that impose an *Eigenvariable condition* on the variable and those which do not. Quantified formulas whose decomposition introduces an *Eigenvariable condition* are of type δ, whereas formulas that introduce freely instantiable unbound variables have type γ. In formulas with free variables we will often indicate the type of the variables by printing the type in superscript (e.g. x^γ).

Signed formulas F can be represented as trees with one node for each subformula of F. This leads to the notion of an *Indexed Formula Tree* (IFT) which can be recursively defined over the structure of F.

δ	δ_0
$(\forall x.F[x])^+$	$F[x/c]^+$
$(\exists x.F[x])^-$	$F[x/c]^-$

γ	γ_0
$(\forall x.F[x])^-$	$F[x/c]^-$
$(\exists x.F[x])^+$	$F[x/c]^+$

Table 3.2: Uniform Notation (Quantifiers)

ϵ	ϵ_1	ϵ_2
$(A \Leftrightarrow B)^-$	A^0	B^0
$(s = t)^-$	s^0	t^0

Table 3.3: Epsilon Rules

Definition 3.1.2 *(Indexed Formula Tree) Let F be a signed formula. The* indexed formula tree *for F is defined as the smallest tree T for which the following holds:*

1. *The root of T is labeled with F.*
2. *If there is a node n in T that is labeled with a signed formula $\alpha(\alpha_1^{p_1}, \alpha_2^{p_2})$ then n has children n_1, n_2 that are labeled with $\alpha_1^{p_1}$ and $\alpha_2^{p_2}$ respectively.*
3. *If there is a node n in T that is labeled with a signed formula $\beta(\beta_1^{p_1}, \beta_2^{p_2})$ then n has childs n_1, n_2 that are labeled with $\beta_1^{p_1}$ and $\beta_2^{p_2}$ respectively.*
4. *If T has a node n that is labeled with a signed formula $\gamma(Qx.F[x])^p$ for $Q \in \{\exists, \forall\}$ then n has finitely many childs $n_1, \ldots, n_m$ that are labeled with $F[c_i/x]^p$ for terms c_i. We say that m is the* multiplicity *of n.*
5. *If T has a node n that is labeled with a signed formula $\delta(Qx.F[x])^p$ for $Q \in \{\exists, \forall\}$ then n has exactly one child with label $F[x^\delta]^p$.*

The above definition only covers IFTs for formulas in $\mathcal{L}$ which is sufficient for this thesis. However, [Aut03] defines IFTs in a way that they can also represent formulas of various modal-logics. We have also left out ζ-type formulas (e.g. positive equivalences) because they do not occur throughout this thesis.

Nodes of the tree are also referred to as *occurrences*, while the formula associated with a node is called the *label* of the occurrence (write *label*(o) for the label of an occurrence o). Because of the close correspondence between a subformula of a formula F and the subtree which represents this formula in the IFT for F, we will only distinguish between a node and its label if this is necessary. In this case we use capital letters O to refer to the label of an occurrence o.

A sample FVIFT for the formula 3.2 is displayed in Figure F 1

$$((M \wedge N \wedge (M \wedge N \Rightarrow P_1)) \Rightarrow^\alpha P)^+ \tag{3.2}$$

In the remainder of this thesis we often have to distinguish between dependent and indepedent nodes in an IFT.

$$\begin{tikzcd}
 & & & (((M \wedge N) \wedge (M \wedge N \Rightarrow P_1)) \Rightarrow^{\alpha} P_2)^{+} \arrow[dl] \arrow[dr] & \\
 & & ((M \wedge N) \wedge^{\alpha} (M \wedge N \Rightarrow P_1))^{-} \arrow[dl] \arrow[dr] & & P_2^{+} \\
 & (M \wedge^{\alpha} N)^{-} \arrow[dl] \arrow[d] & & ((M \wedge N) \Rightarrow^{\beta} P_1)^{-} \arrow[dl] \arrow[dr] & \\
M^{-} & N^{-} & (M \wedge^{\alpha} N)^{+} \arrow[dl] \arrow[dr] & & P_1^{-} \\
 & M^{+} & & N^{+} &
\end{tikzcd}$$

Figure F 1: Indexed formula tree for $(((M \wedge N) \wedge (M \wedge N \Rightarrow P)) \Rightarrow^{\alpha} P)^{+}$. Subscripts of literals are added for later reference.

Definition 3.1.3 *(Dependent Occurrence) Let a be a node in an* IFT *R. We say that a is* dependent *in tree R if there is a node b in R that is β-related to a. Otherwise we call a an* independent occurrence *in R.*

It makes sense to introduce a partial ordering $\prec$ on nodes of an IFT, where $n_1 \prec n_2$ if the length of the path from the root to n_1 is smaller then the distance between the root and n_2; i.e. we have $n_1 \prec n_2$ if n_1 is an ancestor of n_2

Indexed formula trees are used in CORE to represent the quantifier dependencies of a proof state in the following way: when we load a goal formula G together with axioms $Ax_1, \ldots, Ax_n$ from which we want to derive G the system creates an IFT for the signed formula $(Ax_1 \wedge \ldots \wedge Ax_n \Rightarrow G)^{+}$. The system also creates an *free variable indexed formula tree* (FVIFT) for the same formula. FVIFT and IFT together represent a CORE proof state where proof search manipulates the FVIFT while the IFT is used to maintain the dependencies between quantifiers. The FVIFT can be easily obtained from an IFT by simply removing all nodes of type γ and δ.

3.2 Logical Contexts

A look at the sequent calculus derivation of the sample formula 3.2 illustrates how the uniform notation describes the "behavior" of the subformulas during the proof.

As an example consider the literal P_1^{-} in Figure F 1. The negative polarity indicates that P_1 will be found on the left-hand side in all sequents in which it occurs as a sequent formula. When comparing this with the sequent proof in Figure F 2 one easily sees that this is indeed the case. Similarly, one can see that the s-formula $((M \wedge N)^{+} \Rightarrow P^{-})^{-}_{\beta}$ only occurs on the left-hand side of sequents. Decomposition of this formula by the $\Rightarrow_L$ rule splits the sequent into two sequents (as correctly indicated by type β) which contain the subformulas P_1^{-} and $N^{+} \wedge M^{+}$ respectively.

This leads to the notion of a *context*. Intuitively, all formulas that can occur

$$
\dfrac{\dfrac{\dfrac{M,N,\boxed{P_1}\vdash P_2 \qquad \dfrac{M,N\vdash P_2,M \quad M,N\vdash P_2,N}{M,N\vdash P_2,M\wedge N,}\;\wedge_R}{M,N,(M\wedge N\Rightarrow \boxed{P_1})\vdash P_2}\;\Rightarrow_L}{(M\wedge N)\wedge(M\wedge N\Rightarrow \boxed{P_1})\vdash P_2}\;\wedge_L^*}{(M\wedge N)\wedge(M\wedge N\Rightarrow \boxed{P_1})\Rightarrow P_2}\;\Rightarrow_R
$$

Figure F 2: Proof of the sample formula in the sequent calculus. Subscripts refer to literals in F 1

together in a sequent belong to the same context. By using uniform notation the context of a formula can be statically determined from the FVIFT; i.e. two formulas F and G belong to the same context if their *first common ancestor* node is of type α in which case we call F and G α-related. In case there are multiple ancestors $n_1, \ldots, n_k$ the first common ancestor always refers to the maximal t_i with respect to $\prec$ (i.e. to the lowest node in the tree). In the example in Figure F 2 P_1^- and P_2^+ have the first common ancestor $(M \wedge N \wedge (M \wedge N \Rightarrow P)) \Rightarrow^\alpha P)^+$ of type α and hence belong to the same context. However, P_1^- and N^+ belong to different contexts as their first common ancestor $(M \wedge N \Rightarrow^\beta P)^-$ is of type β (i.e. we say they are β-related). Comparing this to the sample proof (Figure F 2), it is easy to verify that the s-formulas N^+ and P_1^- never occur together in a sequent.

3.3 Replacement Rules

In CORE we use formulas that occur in the context of a subformula directly as transformation rules in a proof. As an example we might want to use the implication $(M^+ \wedge N^+ \Rightarrow^\beta P_1^-)^-$ in 3.2 to replace P_2^+ by the formulas M^+ and N^+. This would transform 3.2 into

$$((M^- \wedge^\alpha N^-) \wedge (M^+ \wedge N^+ \Rightarrow^\beta P^-)^- \Rightarrow^\alpha M^+ \wedge^\beta N^+)^+ \qquad (3.3)$$

In a next step we can then use the literals M^- and N^- to transform M^+ and N^+ to $true^+$, which yields the new overall formula

$$((M^- \wedge^\alpha N^-) \wedge (M^+ \wedge N^+ \Rightarrow^\beta P^-)^- \Rightarrow^\alpha true^+ \wedge^\beta true^+)^+ \qquad (3.4)$$

This corresponds to the aforementioned reasoning at the assertion level, which provides a more intuitive basis for interactive proof search. In CORE we use the annotation of signed formulas to generate replacement rules that enable us to perform the transformation steps we described above. The P_1^- in the formula $(M^+ \wedge N^+ \Rightarrow^\beta P_1^-)^-$ is a negative occurrence of the literal P_2^+ in the same context. In a resolution or matrix calculus these literals would correspond to two resolvents or a connection respectively. We can therefore consider formulas that are β-related to P_1^- as subgoals to which we can reduce P_2^+. This reduction to subgoals is facilitated in CORE by the definition of an *admissible replacement rule.*

Definition 3.3.1 *(Admissible Replacement Rules) Let a be an occurrence with polarity p in some* FVIFT *T. Then $i \rightarrow < v_1, \ldots v_n >$ is an* admissible replacement rule *for a, if*

1. *i is α-related to a by a first common ancestor c*

2. *$\{v_1, \ldots, v_n\}$ contains exactly those occurrences that are $\beta-$ related to i and occur below c (i.e. $c \prec v_i$) or, v_1 and i are left- and right-hand side of a negative equation or equivalence and $\{v_2, \ldots, v_n\}$ are all occurrences that are β-related to v_1 and i and occur below c.*

In particular, this means that a positive (negative) occurrence M^+ (M^-) without any β-related occurrences implies that $M^- \rightarrow true^+$ ($M^+ \rightarrow false^-$) is an admissible replacement rule.

Internally the system distinguishes between *resolution replacement rules* and *rewrite replacement rules.* Rewriting replacement rules are all those rules that originate from a (negative) equation or equivalence. All other rules are referred to as resolution replacement rules.

However, not every replacement rule that is admissible for an occurrence a is also applicable on that occurrence. *Applicable rules* are characterized by the following definition

Definition 3.3.2 *(Applicable replacement rules) A replacement rule $\mathcal{R} = i \rightarrow < v_1, \ldots v_n >$ is applicable for an occurrence a, if*

1. *there is a sub-occurrence a' of a such that the label of a' unifies with i under a substitution σ and*

2. *if $\mathcal{R}$ is a resolution replacement rule, then i and a must be of opposite polarities.*

Remark: The definition of admissible replacement rules has the rather unintuitive consequence that a rule like $P^- \rightarrow < M^{p_M}, N^{p_N} >$ can only be used to replace *positive* occurrence with label P^+ by M^{p_M} and N^{p_N}, while intuition would suggest that it replaces an occurrence with negative polarity. When displaying resolution replacement rules we will therefore always invert the polarity if the input occurrence; i.e. the above rule would be displayed as $P^+ \rightarrow < M^{p_M}, N^{p_N} >$. Using this notation we display rules in a style that is common in the field of term-rewriting.

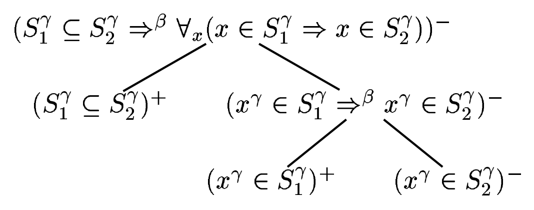

$(S_1^\gamma \subseteq S_2^\gamma \Rightarrow^\beta \forall_x(x \in S_1^\gamma \Rightarrow x \in S_2^\gamma))^-$
$(S_1^\gamma \subseteq S_2^\gamma)^+$
$(x^\gamma \in S_1^\gamma \Rightarrow^\beta x^\gamma \in S_2^\gamma)^-$
$(x^\gamma \in S_1^\gamma)^+$
$(x^\gamma \in S_2^\gamma)^-$

Figure F 3: Free variable indexed formula tree for $\forall S_1, S_2.(S_1 \subseteq S_2 \Rightarrow \forall x.(x \in S_1 \Rightarrow x \in S_2)))^-_\gamma$

3.4 Replacement Rules and Assertion Application

To illustrate the relationship between replacement rules and assertion level reasoning let us consider the assertion (taken from [Hua94])

$$\forall S_1, S_2.(S_1 \subseteq S_2 \Rightarrow \forall x.(x \in S_1 \Rightarrow x \in S_2)) \tag{3.5}$$

and the corresponding FVIFT (see Figure F 3). Because we have to treat 3.5 as an axiom the signed formula in Figure F 3 is of negative polarity. From this tree we can read of all admissible replacement rules that are justified by this tree. These are the following

$$\begin{array}{l}
(x \in S_1)^- \rightarrow < (x \in S_2)^-, (S_1 \subseteq S_2)^+ > \\
(x \in S_2)^+ \rightarrow < (x \in S_1)^+, (S_1 \subseteq S_2)^+ > \\
(S_1 \subseteq S_2)^- \rightarrow < (x \in S_2)^-, (x \in S_1)^- > \\
(S_1 \subseteq S_2)^- \rightarrow < (x \in S_1 \Rightarrow x \in S_2)^- > \\
(x \in S_1 \Rightarrow x \in S_2)^- \rightarrow < (S_1 \subseteq S_2)^- > \\
(S_1 \subseteq S_2 \Rightarrow (x \in S_1 \Rightarrow x \in S_2))^+ \rightarrow \; true^+
\end{array}$$

A careful examination of these rules reveals that they encode all application directions of the assertion 3.5. Take for example the rule

$$(S_1 \subseteq S_2)^- \rightarrow < (x^\gamma \in S_1 \Rightarrow x^\gamma \in S_2)^- >$$

This corresponds exactly to the following rule for the SK (remember that we apply SK-rules bottom-up)

$$\frac{\Delta, \forall x.x \in S_1 \Rightarrow x \in S_2 \vdash \Sigma}{\Delta, S_1 \subseteq S_2 \vdash \Sigma} \tag{3.6}$$

where we replace a formula in the antecedent (in Huang [Hua94] we find a ND version of the rule). In a similar way rule

$$(x \in S_2)^+ \rightarrow < (x \in S_1)^+, (S_1 \subseteq S_2)^+ >$$

corresponds to the opposite application direction of rule 3.6, i.e.

$$\frac{\Delta, S_1 \subseteq S_2 \vdash \Sigma}{\Delta, \forall x. x \in S_1 \Rightarrow x \in S_2 \vdash \Sigma} \tag{3.7}$$

What becomes clear is that replacement rules make it possible to apply assertions in a much more intuitive way as this has to be done in the sequent or natural deduction calculus where we have to decompose an assertion in order to apply it or, alternatively, would have to extend the caclulus by assertion rules (see [Hua94]), such as 3.6 and 3.7, for every assertion.

Based on the definitions of indexed formulas, contexts and replacement rules we can now introduce the CORE calculus.

3.5 The CORE Calculus

A proof state in CORE is represented as a pair $< \varphi, C_v >$, where φ is a FVIFT and C_v is an IFT that represents constraints on the instantiations of variables in φ. We prove a formula φ by transforming the initial proof state $< \varphi, C_v >$ into a proof state $< \varphi', C_v >$ where φ' is either $true^+$ or $false^-$.

CORE provides 12 rules to transform a proof state into exactly one new proof state. This ensures that a proof state can always be represented as a single formula. The rules that we make use of in this thesis are (1) the application of a replacement rule to a subformula, (2) the rule to increase the multiplicity of a variable, (3) the contraction rule and (4) the cut-rule. When we now consider these rules we restrict our attention to the effect that these rule have on the FVIFT φ.

3.5.1 Application of Replacement Rules

In Section 3.3 we defined which replacement rules can be applied to a given subformula (i.e. an occurrence). A question that still needs to be answered is that of how application of a replacement rule $\mathcal{R}$ of form $i^q \rightarrow < v_1^{p_1}, \dots v_n^{p_n} >$ affects the proof state. Let us therefore assume that we want to apply $\mathcal{R}$ to an occurrence a^p of an FVIFT T. In order for the rule to be applicable its input i^q has to unify with a^p under a substitution σ and it must hold that $p \neq q$. If this is the case a^p is replaced by a conjunction of the $v_i^{p_i}\sigma$. This conjunction is expressed by β-relating the $v_i^{p_i}\sigma$. Because we are dealing with first-order logic the rule as well as the subtree below a^p do not contain any elements from a modal logic. The effect of applying the rule is therefore a replacement of a^p by a subtree $\beta(v_1^{p_1}\sigma, \dots, v_n^{p_n}\sigma)$. This effect is shown graphically in Figure F 4. The subtree for Σ indicates that the newly introduced values $v_i\sigma_i$ will have the same context Σ as a^p. The situation is slightly different when we apply a rewriting replacement rule. Rewriting replacement rules originate from negative equivalences or equations such as $(A^0 \Leftrightarrow B^0)^-$ which can be applied in two directions; that is, to positive and negative occurrences. They are special in the sense that their input i as well as $v_1^{p_1}$ are of undefined polarity. When

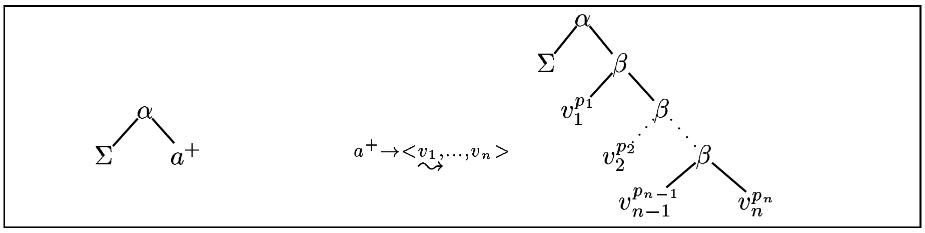


Figure F 4: Application of a resolution replacement rule

applying such a rule the occurrence v_1 inherits the polarity from the application occurrence a^p to which the rule is applied. More precisely, after application of a rewriting replacement rule we have $p_1 = p$

3.5.2 Instantiation of Variables

A problem that has not yet been addressed is that of instantiating variables and the related problem of increasing the multiplicity of variables. We refer to [Aut03] for a detailed treatment of how this is realized in CORE. Here we only sketch how variables are instantiated.

Instantiating a variable x^γ as in $P[x^\gamma]$ is done in CORE by replacing the node of the FVIFT for $P[x^\gamma]$ by a node with the new label $P[t/x^\gamma]$, where t is the term with which x^γ is instantiated. The same procedure is then recursively applied to all subnodes of $P[x^\gamma]$.

Once the x^γ is instantiated in $P[x^\gamma]$ it is not possible to instantiate x^γ in P any further. However, any complete proof system has to be able to instantiate universal variables (γ type variables) more than once. This problem is commonly refered to as "increasing the multiplicity" of variables. Each calculus has its own way to deal with this problem. While in matrix proof search (see for example [Wal90]) one has to guess a "maximal" multiplicity and then copy all literals in advance, the sequent calculus copies quantified formulas, before instantiating them.

In CORE the mulitplicities of individual variables are increased on demand by copying all subformulas that contain these variables and α-relate this copy to the respective source subformula before renaming the variable in the copy. As an example Figure F 5 describes what happens with a subtree for $P(x^\gamma) \wedge Q(x^\gamma)$ if we increase the multiplicity of x^γ.

3.5.3 Contraction

A relatively simple, but yet powerful rule is the so called *contraction rule* which allows to copy subtrees of the FVIFT on demand. Although the need for such a rule might not be immediately clear it will soon turn out that it is important with respect to the task structure to be described in Chapter 4. For the moment the need for such a rule shall only be motivated by a small example: assume

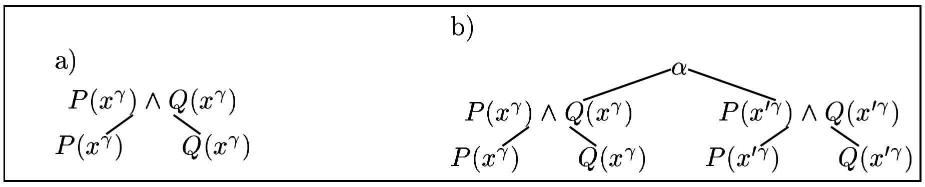

a)
$P(x^\gamma) \wedge Q(x^\gamma)$
$P(x^\gamma)$
$Q(x^\gamma)$
b)
α
$P(x^\gamma) \wedge Q(x^\gamma)$
$P(x'^\gamma) \wedge Q(x'^\gamma)$
$P(x^\gamma)$
$Q(x^\gamma)$
$P(x'^\gamma)$
$Q(x'^\gamma)$

Figure F 5: Increasing the multiplicity of x^γ in $P(x^\gamma) \wedge Q(x^\gamma)$. a) shows the tree before the variables are instantiated and b) after the instantiation was carried out.

one wants to prove the formula

$$(A^- \vee^\beta A^-)^- \Rightarrow^\alpha A^+$$

and is for some reason restricted to rewrite only the right-hand side of the implication (this is important when modelling SK or ND derivations in CORE). Then aplication of $A^+ \to A^-$ which can be generated from any of the A^- yields

$$(A^- \vee^\beta A^-)^- \Rightarrow^\alpha \neg(A^-)^+$$

which cannot be proven any more. However, copying A^+ before application of the rule would have resulted in the formula

$$(A^- \vee^\beta A^-)^- \Rightarrow^\alpha A^+ \vee^\alpha \neg(A^-)^+$$

which can be proven by generation of $A^+ \to \top$ from the rightmost A^- and application of this rule to A^+.

The contraction rule makes it possible to replace a subtree Q_c by a subtree which α-relates Q_c and a copy Q'_c of it; i.e. Q_c becomes replaced by $\alpha(Q_c, Q'_c)$.

3.5.4 Cut Rule

Sometimes one wants to do speculative steps during the process of proving a theorem, like introducing a formula A as a lemma, which is only later verified. Such speculative steps will for instance be of major importance when it comes to application of tactics and methods (see Chapter 6). In CORE, speculative steps are modelled by application of a so called *cut-rule.* The cut-rule in CORE is a mapping of the cut-rule as is used in sequent and natural deduction calculie to the CORE framework. For the sequent calculus, the cut-rule has the following or a similar form (see for example [Gal86]):

$$\frac{\Sigma \vdash A, \Delta \quad \Sigma, A \vdash \Delta}{\Sigma \vdash \Delta}\, Cut(A) \tag{3.8}$$

which makes it possible to introduce arbritrary formulas A into the proof as a sort of lemma. When applying the cut rule one often speaks about "introducing a cut-formula A".

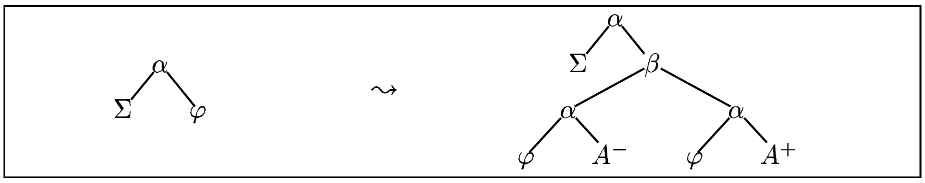


Figure F 6: Introduction of cut-formula A wrt. φ in CORE.

Cut-formulas in CORE have to be introduced with respect to a subformula φ (which corresponds to the Δ in 3.8. Introduction of A over φ is done by replacing the part of the FVIFT for φ by a new subtree for $\beta(\alpha(A^-,\varphi),\alpha(A^+,\varphi))$.

In the resulting tree $\alpha(A^-,\varphi)$ corresponds to the sequent $\Sigma, A \vdash \Delta$ in 3.8 where we can use the cut-formula A to derive Δ. Similarly, the subtree $\alpha(A^+,\varphi)$ corresponds to the sequent $\Sigma \vdash A, \Delta$. Intuitively this can be seen as a new obligation to proof A which we can then safely use as a lemma in the sequent $\Sigma, A \vdash \Delta$.

3.6 Window Inference

On top of the calculus rules decribed above, CORE supports reasoning with a so called *window inference technique* which is based on ideas of [RS93]. In this section we describe this communication layer which makes it possible to focus the reasoning process to subformulas of the overall goal. This can be done by placing a *window* (focus) on a subtree of the FVIFT. As a result, the surroundings of this window are hidden from the user. However, this operation does not alter the proof state but restricts the view on it to a particular subtree. The context of the active window is made available to the user as a list of the replacement rules that are admissible for the content of the window.

Autexier [Aut03] shows that the window inference mechanism can be used to structure proofs in a way that sequent calculus as well as natural deduction proofs can be simulated in CORE.

The window inference meachanism of CORE consists of rules to focus and unfocus certain subformulas as well as window versions of each of the CORE calculus rules. It is worth pointing out that these window inference rules do not extend the reasoning capabilities of CORE. Rather, the window versions of the calculus rules are internally realized purely based upon the CORE calculus rules and the rules for opening and closing windows.

3.6.1 Opening and closing windows

The two basic window rules are those for opening and closing windows on subformulas (subtrees). By applying these operations to an FVIFT we impose a *window structure* on this tree. As a window structure for an FVIFT R we understand a partial function f_R from an enumerable set W of window

$$\begin{tikzcd}[column sep=small]
 & \boxed{(P(x) \Rightarrow^{\alpha} Q(x) \wedge R(x))^{+}} \arrow[dl] \arrow[dr] & & \\
\boxed{P(x)^{-}} & & (Q(x) \wedge^{\beta} R(x))^{+} \arrow[dl] \arrow[dr] & \\
 & Q(x)^{+} & & \boxed{R(x)^{+}}
\end{tikzcd}
\qquad
\begin{tikzcd}[column sep=small]
 & \boxed{(P(x) \Rightarrow^{\alpha} Q(x) \wedge R(x))^{+}} \arrow[dl] \arrow[dr] & & \\
\boxed{P(x)^{-}} & & (Q(x) \wedge^{\beta} R(x))^{+} \arrow[dl] \arrow[dr] & \\
 & Q(x)^{+} & & R(x)^{+}
\end{tikzcd}$$

Figure F 7: Sample window structures for the FVIFT for $(P(x) \Rightarrow Q(x) \wedge R(x))^{+}$

names (e.g. $W = \mathbf{N}$) into the set of subtrees of R. Intuitively, when opening a new window w on a subtree r in R then we add the tuple (w, r) to f_R. Closing a window is exactly the inverse operation which removes a tuple from f_R. We will depict a window structure f_R on an FVIFT R by drawing boxes around subtrees r for which a window w exists (i.e. if $f_R(w) = r$). Figure F 7 (left) shows the FVIFT for $(P(x) \Rightarrow Q(x) \wedge R(x))^{+}$ after opening windows on $(P(x) \Rightarrow Q(x) \wedge R(x))^{+}, P(x)^{+}$ and $R(x)^{+}$ respectively. Figure F 7 (right) shows the same tree with a new window structure where the window on $R(x)$ is closed.

It is useful to impose a partial ordering $\prec_{f_R}$ on windows w, where $w \prec_{f_R} w'$ means that $f(w')$ denotes a subtree of $f(w)$. Windows that are maximal with respect to this ordering are called *active windows*. These are the windows that can be manipulated by application of CORE calculus rules. We are now able to define what the *subwindows* of a window are

Definition 3.6.1 *(Subwindows) If w is a window in an* FVIFT *R then the set of all* subwindows *of w is defined as*

$$Subwindows(w) = \{w' | w \prec w' \textit{ and } w' \textit{ is a window in } R\}$$

Windows can be closed, provided that they do not contain any subwindows. Note that opening and closing of windows never affects the FVIFT, but rather the "view" the user takes on the problem represented by the tree.

3.7 Oracle Rule

When it comes to application of methods we will need to be able to apply abritrary replacement rules (i.e. replacement rules that are not admissible). We can perform such speculative steps with the help of the cut-rule. Assume we want to simulate application of the (non-admissible) replacement rule

$$I \twoheadrightarrow < V_1, \ldots, V_n > \tag{3.9}$$

Note that we use uppper case letters to indicate that I and V_i are formulas, rather than occurrences as in replacement rules. We can simulate application of this rule by performing a cut over $(V_1 \wedge \ldots \wedge V_n)$. This replaces the application occurrence i with label $label(i) = I$ in an FVIFT by the new subtree in Figure F 8. We can see that application of the cut has reduced the goal i to two new subgoals. The subtree $\alpha(\beta(V_1, \ldots, V_n)^+, i)^+$ basically represents the fact that, instead of proving i we can now also proof all of the V_i. This tree can therefore be seen as the result of applying 3.9 to i.

The second subtree $\alpha(\alpha(V_1, \ldots, V_n)^-, i)^+$ encodes the goal to show that I follows from $(V_1 \wedge \ldots \wedge V_n)$. A proof of this subgoal therefore warrants the use of 3.9. Note that 3.9 had been admissible if we have had $((V_1 \wedge \ldots \wedge V_n)^+ \Rightarrow I)^-$ somehwere in the context of i.

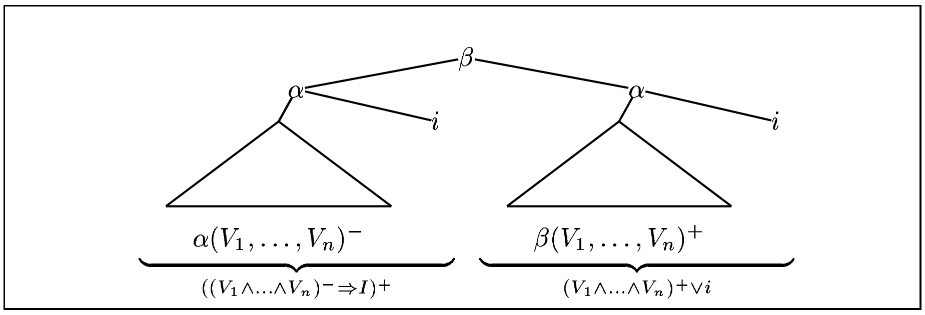

β
α
i
α
i
$\alpha(V_1, \ldots, V_n)^-$
$\beta(V_1, \ldots, V_n)^+$
$((V_1 \wedge \ldots \wedge V_n)^- \Rightarrow I)^+$
$(V_1 \wedge \ldots \wedge V_n)^+ \vee i$

Figure F 8: Oracle application $I \rightarrow< V_1, \ldots, V_n >$.

We can therefore say that application of 3.9 is a ***speculative*** proof step as its validity depends on a proof of the additional goal $((V_1 \wedge \ldots \wedge V_n)^- \Rightarrow I)^+$ CORE provides a calculus rule to apply such speculative proof steps. Rules as in 3.9 are then called *oracle rules*. When we apply such an oracle rule we have to distribute the V_i over two sets, conditions Φ and values V such that $\Phi \cap V = \emptyset$ and $\Phi \cup V = \{V_1, \ldots, V_n\}$. The resulting cut is independent from the distribution of the V_i. However, application of the oracle rule will return two lists of windows. The first list, the so called *condition list* contains windows on those V_i in $\alpha(\beta(V_1, \ldots, V_n)^+, i)^+$ that belong to the condition set Φ. Similarly, the *replacement list* contains windows on the V_i in the value set V. We will make use of this technical detail in Section 6.3. To indicate the distribution of the V_i over Φ and V we frequently display oracle rules in the form $[\Phi]\ I \rightarrow< V_1, \ldots, V_k >$.

3.8 Interactive Theorem Proving with CORE

When using CORE the user currently works with the window inference mechanism. This means that when we invoke CORE in interactive mode on a goal G with axioms $Ax_1, \ldots Ax_n$ it assembles an IFT for the formula $(Ax_1, \ldots Ax_n \Rightarrow G)^+$ and creates an initial window on G which is presented to the user. The content of that window can then be altered by application of the window versions of COREs calculus rules. Typically this will be an application of a replacement rule. Proof search in CORE is therefore characterized by two major kinds of choices.

$$\begin{tikzcd}[column sep=small]
 & & (P(x) \wedge^{\alpha} Q(y) \Rightarrow^{\beta} R(x,y))^{-} \arrow[dl, dash] \arrow[dr, dash] & \\
 & (P(x) \wedge^{\alpha} Q(y))^{+} \arrow[dl, dash] \arrow[dr, dash] & & R(x,y)^{-} \\
P(x)^{+} & & Q(y)^{+} &
\end{tikzcd}$$

$$\begin{array}{l}
R(x,y)^{+} \rightarrow < P(x)^{+}, Q(y)^{+} > \\
R(x,y)^{+} \rightarrow < P(x)^{+} \wedge Q(y)^{+} > \\
P(x)^{-} \rightarrow < Q(y)^{+}, R(x,y)^{-} > \\
Q(y)^{-} \rightarrow < P(x)^{+}, R(x,y)^{-} > \\
P(x) \wedge Q(x) \rightarrow R(x,y)
\end{array}$$

Figure F 9: IFT with replacement rules that can be generated from this tree.

1. The first kind of choice consists of the selection of a subformula in the active window on which the user wants to focus the proof search. This kind of choice will be referred to as *foci-choice.*

2. Having focussed the proof search on a window, one of the admissible replacement rules for that window has to be selected for application. This will be referred to as *rule-choice*

However, although COREs inference mechanism is in principle well suited for interactive proof construction, foci– and rule choice still imply some difficulties. Two problems are particulary salient.

3.8.1 Rule Selection

The first of these problems lies in the selection of the appropriate replacement rule for the manipulation of a window. Typically there are many admissible replacement rules for every window that one encounters during a proof. Even for rather trivial problems there are already dozens of admissible rules. The reason for this is twofold.

First, every formula in the context of a window gives rise to the generation of multiple replacement rules. Intuitively this is due to the fact that there is a replacement rule for every possible application direction of this formula (if we see the formulas in the context as a form of assertions). Assume for example that the FVIFT in Figure F 9 was available in the context of a window W. How many admissible replacement rules for W can we obtain from this tree? It is easy to see that each occurrence of this FVIFT can serve as an input for at least one replacement rule. Because the tree has 5 nodes we obtain 5 replacement rules that are admissible in W. This illustrates that for each subformula in the context of a window we can hence generate at least as many replacement rules as there are nodes in the tree that represents this formula. Second, the problem is made worse by the fact that the system currently automatically loads some lemmas such as the transitivity of $=$ and the induction axiom which are then contained in every FVIFT. Because these lemmas will be found in the context of every window in a proof, they will cause generation of even more replacement rules.

Furthermore, as was noted earlier, not every admissible replacement rule is also applicable. When selecting a replacement rule for application the user first has to identify the applicable rules amongst all admissible rules. Determining whether a rule is applicabe means checking whether the input of the rule unifies with a subformula of the current window. Having identified these applicable rules it remains to select one of these rules for application along with a subformula of the focus to which the formula is to be applied.

It is easy to imagine that this constitutes quite a challenging task when too many admissible replacement rules are available for a given window.

3.8.2 Foci Choice

A further difficulty for proof construction arises from the need to identify subformulas on which the proof search can be focussed. Because each proof state in CORE simultaeneously contains all subgoals together with their alternatives it is necessary to identify an apropriate set of subgoals, on which the proof search can then be focussed.

The two problems outlined in 3.8.1 and 3.8.2 above motivate the need for

- a mechanism that aids the user in structuring the proof search by preventing him from focussing on arbritrary subformulas.
- a suggestion mechanism that is able to check all replacement rules and tactics for applicability in a given proof state and makes suggestions to the user about which rule to apply.

In the next chapter we will introduce the notion of a *task* datastructure which can be used to structure the search for proofs. Later we will see how the multi-agent mechanism ΩANTS that is currently used as a suggestion mechanism for the ΩMEGA theorem prover was adapted to the CORE system to perform exactly the tasks described above. However, before being able to make proper use of the mechanism we have to introduce a means to structure the search for proofs by systematically focussing on certain subformulas.

3.9 Chapter Summary

In this chapter we have seen how CORE uses annotated formulas to realize a contextual reasoning style. We have also demonstrated how information in the logical context of a formula can be directly used in a proof in the form of replacement rules, which offers a convienient way to apply assertions directly. Finally, we pointed to some inconviniences of interactive proof construction in CORE. In the next chapter we will introduce a further communication layer on top of the window inference mechanism which tries to overcome some of these problems.

Chapter 4

Tasks – Organizing Proof Search

In the previous section we saw how CORE s window inference technique is used in interactive proof construction. However, we also pointed to some difficulties this still implies for interactive proof construction. As the main problem we identified the fact that the context of a window can only be presented to the user as a (usually long) list of replacement rules. The disadvantage of this presentation is that although the user often knows which assertion he wants to apply to the current window, he might not be able to easily identify the replacement rule that corresponds to the appropriate application direction of the assertion. Let us therefore think about how the representation of a window can be improved to ease the application of an assertion. Assume that we have a goal-formula of the form $A \Rightarrow B \wedge C$, together with the axioms $Ax_1, \ldots, Ax_n$. Let us further assume that the assertion $\forall x.Q[x] \Rightarrow B$ is among these axioms. A convenient way of presenting this situation to the user would be to list the axioms one in each line followed by the goal formula. This can be schematically depicted as

$$\begin{bmatrix} Ax_1 \\ \vdots \\ (Q[x^\gamma] \Rightarrow B)^- \\ \vdots \\ Ax_n \end{bmatrix} \tag{4.1}$$
$$(A \Rightarrow B \wedge C)^+$$

Using this representation the formulas in square brackets represent all the information that is available to infer the goal $(A \Rightarrow B \wedge C)^+$. The presentation we have in mind is therefore related to the one used for proof presentation in the THEOREMA system [PB02]. How might a user want to proceed in this situation? It seems likely that he would continue by trying to prove the goal $(B \wedge C)^+$ with the help of the additional hypothesis A. In fact, this would result in the new proof situation

$$\begin{array}{c} \left[\begin{array}{c} Ax_1 \\ \vdots \\ (Q[x]^{\gamma} \Rightarrow B)^- \\ \cdot \\ Ax_n \\ A \end{array}\right] \\ \\ (B \wedge C)^+ \end{array} \tag{4.2}$$

In a similar fashion the user might then concentrate on proving B^+, before coming back to establishing C^+. This corresponds to a decomposition of the goal $(B \wedge C)^+$ into two subgoals.

$$\left(\begin{array}{c} \left[\begin{array}{c} Ax_1 \\ \vdots \\ (Q[x^{\gamma}] \Rightarrow B)^- \\ \cdot \\ Ax_n \\ A \end{array}\right] \\ \\ B^+ \end{array} , \begin{array}{c} \left[\begin{array}{c} Ax_1 \\ \vdots \\ (Q[x^{\gamma}] \Rightarrow B)^- \\ \cdot \\ Ax_n \\ A \end{array}\right] \\ \\ C^+ \end{array} \right) \tag{4.3}$$

The user is then confronted with the first element of the list in 4.3. The second element represents the (now inactive) subgoal C^+ which has to be tackled after B^+ is proven. A step into the direction of proving B^+ might be to apply the assertion $(Q[x^{\gamma}] \Rightarrow B)^-$ to B^+ and refine the goal B^+ to $Q[x^{\gamma}]^+$. We would then obtain

$$\left(\begin{array}{c} \left[\begin{array}{c} Ax_1 \\ \vdots \\ (Q[x^{\gamma}] \Rightarrow B)^- \\ \cdot \\ Ax_n \\ A \end{array}\right] \\ \\ (Q[x^{\gamma}])^+ \end{array} , \begin{array}{c} \left[\begin{array}{c} Ax_1 \\ \vdots \\ (Q[x^{\gamma}] \Rightarrow B)^- \\ \cdot \\ Ax_n \\ A \end{array}\right] \\ \\ C^+ \end{array} \right) \tag{4.4}$$

In this chapter we will develop a datastructure of so called *tasks* to be able to present a window together with the assertions in the context of the window as we did in 4.1– 4.4. Furthermore, we will define a set of rules on this datastructure to actually perform the transformations described above. The rules that operate on this datastructure will be entirely realized through application of COREs window inference rules; e.g. rules that realize steps as from 4.1 to 4.2 will go back to the rules for opening new subwindows. Rules that realize the application of a hypothesis as from 4.3 to 4.4 will basically consist of the application of a window replacement rule. In this sense the datastructure devised here introduces a new layer on top of the window inference mechanism on which the user interacts with CORE.

When developing the task datastructure we will pursue the additional aim to employ tasks also as a datastructure for proof planning. The idea is that tasks provide a common interface for automated and interactive reasoning processes. This ideally enables us to tackle inactive subgoals (such as the C^+ in 4.3) with an automated proof planner or an automated theorem prover (ATP) in the background, while the user tries to construct a proof for another subgoal interactively.

The task datastructure that is developed in this chapter will resemble that of a *task* in ΩMEGAs proof planner MULTI [Mei03] and also have much in common with the *focus* datastructure that is used by ΩANTS (see [Sor01]).

4.1 A Calculus for Tasks

In this section we develop a task datastructure on top of COREs window inference technique. Intuitively, the tasks that will be introduced below denote subgoals together with all the formulas that can be used to close this subgoal (all subtrees that are α-related to the goal). Accordingly a task will simply be defined as a list of windows that all occur in the same context.

4.1.1 Definition of Tasks

For proof planning it is often useful to attach a certain role to a window which occurs in a task, e.g. it is often convenient to distinguish between windows that denote axioms and those that would denote a local hypothesis in a ND proof. For instance in 4.2 we could label A as a locally introduced hypothesis, while all other windows inside the brackets are classified as axioms or definitions. This information can be exploited in proof planning and proof presentation. To be able to do so, we first define the notion of a *role* and an *annotated window* on which the definition of tasks will then be based.

Definition 4.1.1 *(Roles)*
We fix a finite set $\mathbf{R} = \{Axiom, Definition, Hyp, \ldots\}$ *consisting of unique role specifiers.*

Definition 4.1.2 *(Annotated Window) An* annotated window A *is a tuple* $A = (W, R)$*, where* W *is a* CORE *window and* $R \in \mathbf{R}$ *is the* role *of* W*.*

In the following we will refer to annotated windows simply as windows. Tasks can now be defined to be sets of annotated windows. To reflect the distinction between axioms (i.e. everything that occurs in brackets in 4.1) and the goal that we want to derive we will divide the set of windows of a task into a *goal window* and *support windows.*

However, before we make this intuition formal we transfer the notion of a dependent occurrence to windows.

Definition 4.1.3 *(Conditional Window) Let w be a window on a subformula F in an* FVIFT *R. We say that the window w is* conditional in R *iff the node in R that is labeled with F is dependent in R. Otherwise we call w* unconditional in R.

Definition 4.1.4 *(Tasks) Let R be the* FVIFT *of the current proof state. A task T is a set of annotated windows $T = \{w_1, \dots, w_n\}$ for R with exactly one* goal window w_i, *$i \in \{1, \dots, n\}$ and* support windows $\{w_1, \dots, w_n\} \backslash \{w_i\}$, *where the following holds if R' is the smallest subtree in R that contains all windows in T:*

1. *the subtrees denoted by the $w_1, \dots, w_n$ are α-related between each other.*

2. *all support windows of T are unconditional in R',*

We denote tasks $T = \{w_1, \dots, w_n, g\}$ with goal window g as $w_1, \dots, w_n \rhd g$ or $\Sigma \rhd g$ if we are not interested in the exact nature of the support windows. Note that this notation is only slightly different to the representation we used at the beginning of the chapter where we displayed the support windows above the goal window.

Selecting one window as the goal window plays a role when reasoning interactively. However, the distinction is not a principle one, something that is reflected by the fact that we will introduce a *shift* rule that allows us to swap a support window with the goal windows.

Here we find an important difference to the sequents in the sequent calculus. In the sequent calculus it is relevant on which side of the sequent an s-formula occurs. In CORE this information is already encoded in the polarities of each formula. We can therefore freely exchange the order of windows in a task. This also motivates our decision to define a task as a *set* of windows. Also note that a task $\Sigma, a^p \rhd a^q$ does not necessarily correspond to an initial sequent in the sequent calculus because the a's might have the same polarity (i.e. $p = q$).

The constraint that no support window of a task must be conditional is important because the content of support windows will be presented to the user as some directly available "knowledge" that can be used to derive the formula in the goal window of the task. If the content of the support windows would be β-related to subtrees that lie outside the respective window, then these trees would automatically become conditions for any replacement rule that is generated from this window; i.e. the β-related subtrees would represent implicit "knowledge" which will be introduced in form of new proof obligations. This is unwanted for interactive proof construction, as the user might want to be able to see whether certain formulas in the "context" are dependent on further formulas, before applying them in the form of a replacement rule.

As an example consider the formula $(D^+ \Rightarrow^\beta (s = t)^-)^- \Rightarrow^\alpha B[s]^+$. Without the requirement that support windows must be unconditional we could generate the following task for the above formula: $(s = t)^- \rhd B[s]^+$. However, although this task gives the impression that the equation $s = t$ could be used directly to transform $B[s]$ to $B[t]$, this is not the case, as $s = t$ is dependent on

D^+ and hence the replacement rule $s \to < t, D^+ >$, instead of $s \to t$ has to be used to carry out the transformation.

Because tasks are basically representations of subgoals we next define when a task is closed.

Definition 4.1.5 *(Closed task) A task $\Sigma \rhd G$ is* closed *iff there exists a $w \in \Sigma \cup \{G\}$ such that w denotes a proved subtree; i.e. w is either $true^+$ or $false^-$.*

An initial problem of deriving a goal G from the axioms $Ax_1, \dots, Ax_n$ is represented by the system as an IFT for the signed formula $(Ax_1, \dots, Ax_n \Rightarrow G)^+$. When reasoning in the just defined task structure, each proof starts out with an initial task which contains a window for the goal formula G as the goal formula and one window for each of the axioms as supports. This motivates the following definition of an *initial task.*

Definition 4.1.6 *(Initial Task) Let G be a formula and $Ax_1, \dots, Ax_n$ formulas that represent axioms from which G can be derived. Let further R be an IFT for $(Ax_1, \dots, Ax_n \Rightarrow G)^+$, w_i a window on a subtree for Ax_i and g a window on G then $w_1, \dots, w_n \rhd g$ is the* initial task *for G.*

From now on we will not distinguish anymore between a window and the formula it contains when we represent tasks. Hence, the initial window for G with axioms Ax_i will be represented as $Ax_1, \dots, Ax_n \rhd G$. Note that this implies that we can encounter tasks of the form $\Sigma, A^p, A^p \rhd G$. In this case the A^p are syntactically equal formulas that occur in different windows. However, because we are dealing with windows, rather than formulas we have to treat the A^ps as different entities.

We have seen that a task with a goal window of type β can be split into two tasks by decomposition of the goal formula (cf. 4.3). Because we always want to keep track about all tasks that are created during a proof attempt we define the concept of an *agenda.*

Definition 4.1.7 *(Agenda) An* agenda *is a set of tasks. An* initial agenda *is an agenda that contains only the initial task for a goal G.*

Tasks on the agenda can be manipulated by decomposition of the goal window (α- or β-decomposition), application of a replacement- or oracle rule to the goal formula, closure of the window on the goal formula or selection of a different goal window (*shift*). In the following section we will define a set of rules that allow us to perform exactly these manipulations. The rules will be of form $rule : TASKS \to 2^{TASKS}$. That is, by application of a rule to a task this task is replaced by zero or more tasks on the agenda. The rules are shown in Figure F 10 in a declarative notation which resembles that of sequent calculus rules. Premises of the rules consist of one or more tasks and possibly additional sideconditions. The conclusion of a rule contains all tasks that replace the premise on the agenda. In that sense these rules can only be applied in forward direction. We now investigate each of the rules in turn.

$$\frac{\Sigma \rhd \alpha(A^{p_A}, B^{p_B})}{\Sigma, B^{p_B} \rhd A^{p_A}}\ \alpha_R \qquad \frac{\Sigma \rhd \alpha(A^{p_A}, B^{p_B})}{\Sigma, A^{p_A} \rhd B^{p_B}}\ \alpha_L$$

$$\frac{\Sigma \rhd \alpha((\neg(A^{-p}))^{p})}{\Sigma \rhd A^{-p}}\ \alpha_\neg$$

$$\frac{\Sigma \rhd \beta(A^{p_A}, B^{p_B})}{\Sigma \rhd A^{p_A} \qquad \Sigma \rhd B^{p_B}}\ \beta$$

$$\frac{\Sigma \rhd G \quad G' = Parent(G)}{\Sigma \rhd G' \qquad Subwindows(G') = \emptyset}\ Focus - Close$$

$$\frac{\Sigma, F \rhd G}{\Sigma, G \rhd F}\ shift \qquad \frac{\Sigma \rhd \diamond}{\emptyset}\ close$$

$$\frac{\Sigma \rhd i}{\Sigma, i \rhd v_1 \ldots \Sigma, i \rhd v_n}\ apply(i \rightarrow < v_1, \ldots, v_n >)$$

Figure F 10: The task manipulation rules.

4.1.2 α-decomposition

Decomposition of a goal formula of type α is the simplest of our task manipulation rules. It realizes the task transformation described in 4.1–4.2. To decompose a formula $\alpha(A^{p_A}, B^{p_B})$ it is merely necessary to open new windows on the A^{p_A} and B^{p_B} and then decide which of the new windows becomes the new goal window. To be able to select either subwindow as new goal window we define three rules:

$$\frac{\Sigma \rhd \alpha(A^{p_A}, B^{p_B})}{\Sigma, B^{p_B} \rhd A^{p_A}}\ \alpha_R \qquad \frac{\Sigma \rhd \alpha(A^{p_A}, B^{p_B})}{\Sigma, A^{p_A} \rhd B^{p_B}}\ \alpha_L \tag{4.5}$$

$$\frac{\Sigma \rhd \alpha((\neg A)^{p})}{\Sigma \rhd A^{-p}}\ \alpha_\neg \tag{4.6}$$

We can see that the decomposition step from 4.1– 4.2 can be realized with rule α_L.

4.1.3 β-decomposition

The rule for decomposition of a goal formula of type β enables us to perform proof steps such as the one from task 4.2–4.3. The rule as it will be defined here is in principle easy. Decomposition of a task with a goal formula $\beta(A^{p_A}, B^{p_B})$ will lead to a split of the task into two tasks with goal formulas A^{p_a} and B^{p_B} respectively. Accordingly, the rule for β-decomposition looks as follows:

$$\frac{\Sigma \rhd \beta(A^{p_A}, B^{p_B})}{\Sigma \rhd A^{p_A} \qquad \Sigma \rhd B^{p_B}}\ \beta \tag{4.7}$$

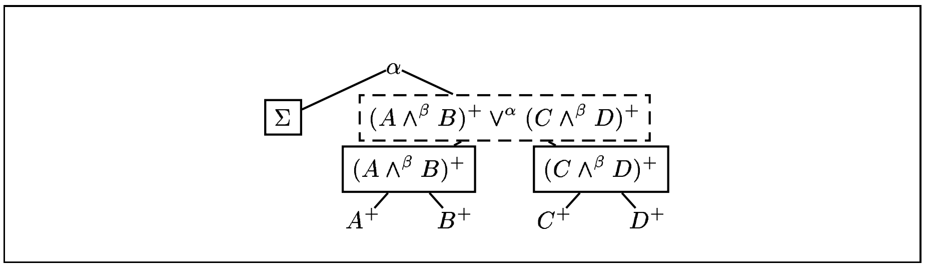

α
Σ
$(A \wedge^{\beta} B)^{+} \vee^{\alpha} (C \wedge^{\beta} D)^{+}$
$(A \wedge^{\beta} B)^{+}$
$(C \wedge^{\beta} D)^{+}$
A^{+}
B^{+}
C^{+}
D^{+}

Figure F 11: FVIFT for the task $\Sigma, (A \wedge B)^{+} \rhd (C \wedge D)^{+}$. Dashed lines indicate existing but inactive windows.

To allow for an unrestricted decomposition of β-formulas it needs a little more effort when implementing this rule than was the case with the rules for α-decomposition. The reason is that there must be no conditional support windows in a task. That is, after decomposition of a goal formula $\beta(G_1, G_2)$ the constituents G_1 and G_2 can only become support windows if we make them unconditional when applying the β-decomposition. This can be done by splitting up the goal formula $\beta(G_1, G_2)$ while retaining the context φ around it. We can achieve this through application of a rule of the form $\varphi(\beta(A, B)) \rightarrow \beta(\varphi(A), \varphi(B))$ which is described in [Sch77] and[Aut03]. Autexier [Aut03] can show that this *Schütte Rule* is admissible in any CORE proof state.

To see why we need the *Schütte Rule*, consider a task $\Sigma, (A \wedge B)^{+} \rhd (C \wedge D)^{+}$. An FVIFT for this task is shown in Figure F 11. If we implement the β-decomposition rule with the help of the *Schütte Rule* we not only change the window structure of the FVIFT as we do with the α-decomposition rule, but we also change the FVIFT itself. For instance, if we β-decompose the goal window $(C \wedge D)^{+}$ we obtain the new FVIFT in Figure F 12.

We see that the task $\Sigma, (A \wedge B)^{+} \rhd C^{+}$ corresponds to the minimal FVIFT R_1 with root-node α_1 and task $\Sigma, (A \wedge B)^{+} \rhd D^{+}$ is represented by the minimal FVIFT T_2 below α_2 (cf. Figure F 12). It is important to note that C^{+} and D^{+} are unconditional in the respective trees R_1 and R_2. Hence, we can easily make them to support windows of a task. This will be important in Chapter 4.1.6 when we define the *shift*-rule.

4.1.4 Larger Decomposition Steps

α- and β-decomposition steps can be combined in a macro-rule that allows us to focus directly on a particular subformula inside the goal window of a task. The uniform types of the nodes in an FVIFT that occur on a path between the selected subformula and the root of the goal window uniquely define a sequence of α- and β-decomposition steps that need to be applied in order to obtain the chosen formula as a goal window in a single step. Such a macro-rule gives us great freedom in the selection of subwindows but simultaneously keeps track about parallel subgoals that appear as tasks on the agenda.

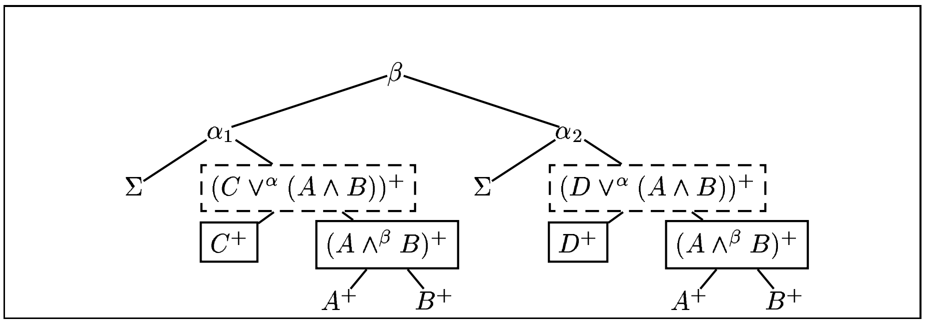


Figure F 12: A logical equivalent FVIFT to the one in Figure F 11. This tree would result from the decomposition of the tree in Figure F 11 if we realize the β-decomposition with the help of the *Schütte Rule.* The left subtree below α_1 corresponds to task $\Sigma, (A \wedge B)^+ \rhd C^+$ while the subtree below α_2 represents task $\Sigma, (A \wedge B)^+ \rhd D^+$.

4.1.5 Focus Close

It is often necessary to undo a decomposition of the goal window of a task, which has to be realized by closing the focus of the goal window G and with it, all other foci below the parent of G. The following *focus-close* rule provides exactly this functionality. Furthermore, this rule is also necessary to realize the sideconditions of other rules (e.g. *shift*, see Section 4.1.6), which require to close windows.

$$\frac{\Sigma \rhd G \quad G' = Parent(G)}{\Sigma \rhd G' \qquad Subwindows(G') = \emptyset}\ Focus - Close \tag{4.8}$$

Unfortunately, it is not enough to simply close all subwindows below the parent-window G' of the window G to which the rule is applied. Because the active subwindows are constituents of other tasks it is necessary to update the tasks structure on the agenda while closing subwindows of G'. Closing subwindows recursively while updating the agenda at the same time can be done relatively straightforward. We have to treat α- and β-windows differently.

1. Windows W of type β with subwindows W_1, W_2 can be simply closed by removing all tasks from the agenda that contain either W_1 or W_2 as goal window. In fact, W_1 and W_2 can only occur as goal windows, because they are mutually β related and can hence not occur as support window. Having removed the respective tasks (if there are any), W_1 and W_2 can be closed (recursion) and a new task with goal window W and the same supports as W_1 and W_2 have to be pushed on the agenda.

2. Closing windows W of type α with subwindows W_1, W_2 is only little more complicated. We can close W by first retrieving the set T of all tasks from the agenda that contain both windows (in fact each task contains either both or none of W_1, W_2). In a second step the close rule is applied

recursively to W_1 and W_2 before W_1 and W_2 are replaced by W in all tasks.

4.1.6 The *shift* rule

The shift rule changes the goal formula of a task. This is particularly important because the rules defined here only allow for manipulation of goal windows. Consider for instance a situation where we have a window on a formula A^p and a window for $(A \Leftrightarrow B)^-$ amongst the support windows for a goal G; i.e. $\Sigma, (A \Leftrightarrow B)^-, A^p \rhd G$. If we now want to apply $A \Leftrightarrow B$ in a forward step to A^p we have to make A^p the goal window. To be able to perform this transformation of a task we define the *shift*-rule.

$$\frac{\Sigma, A \rhd G}{\Sigma, G \rhd A}\ shift$$

Because we realized β-decomposition with the *Schütte-Rule* we can be certain that the goal windows of a task are always unconditional which is a prerequisite for the above definition of the *shift*-rule.

4.1.7 Closing tasks

Of course, there must be a way to remove tasks from the agenda in case they are closed (i.e. one of the windows in the task is proved). This can be done with the following rule for each $\diamond \in \{true^+, false^-\}$.

$$\frac{\Sigma \rhd \diamond}{\emptyset}\ close \tag{4.9}$$

which simply removes $\Sigma \rhd \diamond$ from the agenda.

To be able to make use of COREs reasoning capabilities it is necessary to augment the set of rules by additional ones for the application of replacement and oracle rules to tasks. This is particularly important, because method application will be realized through oracle rule application as will be elaborated in detail in Chapter 6. Here we introduce the task rule for replacement rule application.

4.1.8 Replacement Rule Application

The way tasks are defined, all replacement rules that are admissible for a goal window can be generated from the support windows of a task. When we now define a rule that allows us to apply a replacement rule to a task we have to make sure that we lose no "information" when we apply the replacement-rule. To see what is meant consider a task of the form

$$\Sigma, G^+, (A^+ \Rightarrow^\beta B^-)^- \rhd A^-$$

The window on $(A^+ \Rightarrow^\beta B^-)^-$ justifies the admissible replacement rule $A^- \rightarrow B^-$. Intuitively, application of the replacement rule to A^- should yield an additional window on B^- such that the task that results from application of this rule is $\Sigma, A^-, G^+, (A^+ \Rightarrow^\beta B^-)^- \rhd B^-$; i.e application of the rule should not remove A^- from the task. However, merely applying this rule to A^- would replace A^- by B^- (see 3.5.1), which would yield $\Sigma, G^+, (A^+ \Rightarrow^\beta B^-)^- \rhd B^-$. This is not quite what we want.

The solution to this problem lies in the application of the contraction rule to the goal window. By copying the window before application of the replacement rule, we ensure that no knowledge is "lost". Accordingly, the rule for application of a replacement rule $\mathcal{R} = i \rightarrow < v_1, \ldots, v_n >$ looks as follows

$$\frac{\Sigma \rhd i}{\Sigma, i \rhd v_1 \ldots \Sigma, i \rhd v_n} \; apply(\mathcal{R}) \tag{4.10}$$

The way this rule is defined it can only be applied to an occurrence that is the only content of the goal window of a task; but not to occurrences i inside a goal window (i.e. $G[i]$). However, this is no problem as we can always make an i in $G[i]$ the goal window by application of the macro-rule for α- and β-decomposition (see Section 4.1.4).

4.2 Tasks vs. Sequents and ND

When looking at the effects that α- and β-decompositions have on the task structure there are some similarities to the sequent and natural deduction calculus. The similarities are particularly striking when we compare the motivating example at the outset of the chapter with the "corresponding" proof steps in the ND calculus. In this part of the chapter we will relate the task approach introduced above to the sequent and natural deduction calculus to work out the differences and commonalities.

Although the effects of the α- and β- decomposition rules we have introduced above resemble some of the sequent calculus (i.e. $\wedge_L, \wedge_R, \vee_L, \vee_R, \Rightarrow_L, \Rightarrow_R$) and natural deduction calculus rules (e.g. $\Rightarrow_I, \Rightarrow_E$), there is an important difference. While the rules in SK and natural deduction lead to a real decomposition of the formula they are applied to, the α-decomposition rule merely affect the window structure on the FVIFT. Accordingly, when we α-decompose the goal window of a task this only changes the view we take on the proof state, but not the proof state (FVIFT) itself which modifies the IFT. Consider for example the ND rule

$$\frac{\begin{array}{c}[F]\\ \vdots\\ G\end{array}}{F \Rightarrow G} \; \Rightarrow_I$$

Backward application of this rule decomposes a formula $F \Rightarrow G$ into formulas F and G. The corresponding task rule is the rule for α-decomposition which, applied to a window $(F \Rightarrow G)^+$, merely opens a new subwindow on F and G. We can therefore say, that α-decomposition in CORE only anticipates the corresponding steps in the SK and natural deduction calculus and can therefore be seen as a kind of look-ahead on α-decomposition in these calculi.

It is furthermore worth pointing out that the task structure restricts the set of possible proofs of a theorem. The reason for this is that using tasks we can only open and close windows in a systematic way. This restricts the set of subformulas on which we can focus. Using the task structure therefore imposes a certain structure on proofs. Autexier [Aut03] shows that when we restrict the choice for focusing and replacement-rule application even further it is possible to emulate SK derivations in CORE.

However, the task structure developed here still enables a more flexible reasoning then does the sequent calculus. For instance, when using the macro-rule for α- and β-decomposition we can perform larger decomposition steps than are possible in the SK.

4.3 Tasks in Interactive Proof Search

The task structure that we introduced in this chapter was developed with respect to two aims. Firstly, we wanted to be able to present the formulas contained in the context of a window in a more accessible way then merely as a (possibly long) list of replacement rules. Secondly, while focusing on subformulas during a proof, we wanted to be able to determine conjunctive subgoals outside the current focus. This for instance makes it possible to call automated theorem provers which are encapsulated as concurrent processing threads on these inactive goals while the user is working interactively on the active goal. We will argue that both aims have been fulfilled with the implementation of the task structure.

With respect to the issue of presentation, tasks allow us to present a proof situation to the user in an intuitive way. Instead of listing all replacement rules that can be generated from the context, the support windows give us access to the assertions which we can list line by line on top of the screen, followed by the goal window at the bottom (Figure F 23, Appendix). Such a presentation of a task is meant to resemble more the presentation of proofs in mathematical textbooks, where all available assertions (axioms and definitions, etc) are listed before the theorem is stated. Individual proof steps on the assertion level are then generally justified by reference to these assertions.

In a system state where the user is shown the current task he has the following options to continue the proof. He can either α- or β-decompose the goal window. A second choice is the application of one of the assertions that are represented by the support windows. This can be achieved by simply clicking on the respective formula. The system then presents all replacement rules that can be generated from the chosen support window. It is then the choice of the user to identify the replacement rule that encodes the preferred application direction of the assertion. This is already an improvement to the situation where the user was presented with all replacement rules that are admissible with respect to the current window; i.e. the replacement rules generated from *all* formulas in the context.

However, it is possible to do even better. For many of the rules that can be generated from a single support window in a task, one can see in advance that they are not applicable in the goal window. For other rules it is likely that they do not represent a suitable application of the formula from which they were generated. As a consequence, the system currently uses a heuristic to generate only those rules that are likely to represent the intended application direction of the assertion (see Chapter 5).

In case the user does not know immediately which assertion to apply, he has to fall back on selection of a promising replacement rule that can be generated from one of the support windows. Later we will see how the user is supported in this task by the agent-based suggestion mechanism ΩANTS which identifies applicable replacement rules and presents the most promising of these rules to the user. Because each applicable replacement rule corresponds to an applicable support window we could even exploit this information to highlight applicable support windows on the interface.

A further reason for the introduction of tasks was to be able to determine conjunctive goals outside the current focus to which external theorem provers can be applied in the background as concurrent processes. Tasks also lay the foundation for this aim. The tasks that are created by application of the rules 4.5–4.10 are constantly maintained on the agenda. The agenda therefore represents a segmentation of a proof into different subgoals. In Chapter 8 we will see how the agent architecture ΩANTS can be used to employ external reasoning systems in the background to tackle problems encoded by inactive tasks.

4.4 Chapter Summary

In this chapter we have introduced the task data-structure that allows us to display a window together with its context in an intuitive way to the user. Furthermore, we have developed a set of rules to manipulate tasks; e.g. to perform decomposition steps on the formulas inside particular windows of a task or to apply replacement rules to a task. By reasoning with this task manipulation rules we automatically keep track about parallel subgoals which will show up as separate tasks on the agenda. This provides the basis for the employment of external reasoning systems that try to solve inactive subgoals automatically in the background of the interactive proof process. In the next chapter we will have a look at how interactive proof construction proceeds when we reason with the task structure.

Chapter 5

Tasks - A worked example

In this chapter we make use of the task structure to step through a proof of a simple theorem. In doing so we are able to point out the advantages that the task structure brings for interactive theorem proving as well as putative weaknesses of the approach. In particular we outline the two major strengths of the task structure. The first being the intuitive presentation of subgoals together with the axioms and hypothesis that are available in the context of this subgoal. Secondly, we show how tasks successfully excerpt COREs potential to ease the application of assertions.

The example theorem that we will consider is the following

Theorem 5.0.1 *Let A and B be sets, such that $A \subseteq B$ then it also holds that $2^A \subseteq 2^B$.*

Let us further assume that we can make use of the axioms 5.1– 5.2. Note that so far CORE does not provide a way to define concepts other than expressing them as equivalences or implications. Furthermore, to ease the presentation in this chapter, we assume that the definitions below are encoded as higher-order formulas with polymorphic types [1].

$$\forall M \forall N. M \subseteq N \Leftrightarrow \forall z. z \in M \Rightarrow z \in N \tag{5.1}$$
$$\forall X \forall M. X \in 2^M \Leftrightarrow X \subseteq M \tag{5.2}$$

For the presentation of the proof we assume that a task is presented to the user in a way similar to how we depicted tasks at the beginning of the previous chapter. This means that each support window is displayed in one line and the content of the goal window is shown below the supports. However, as a slight modification we decide to display positive support windows only if they are not the copy of a goal window that was made before the application of a replacement rule (cf. Rule 4.10). This decision is motivated by the observation that we rarely need to use these positive copies. However, we do not get entirely

[1] In the current implementation they still have to be encoded in a sorted first-order logic.

rid of these support windows as they might be useful for backtracking (i.e. they can serve as possible points to which the user might want to backtrack) and in refutation proofs. The copies of positive windows are therefore only hidden but can be made available upon request. Clearly the issue of how to present a task to the user is also related to the development of an adequate graphical user interface for CORE. It should be possible for the user to choose between different presentation modes, depending on how he wants to construct the proof.

Looking at figure F 23 (Appendix) we can see that the presentation of tasks we use in this chapter is very similar to the way tasks are actually presented in the current GUI of our system. The sole difference is that the GUI does not yet indicate which polarity goal- and support windows have. However, this can easily be changed; for instance by coloring each formula inside a task according to its polarity.

$$\forall A, B. A \subseteq B \Rightarrow 2^A \subseteq 2^B \tag{5.3}$$

We can now turn to a proof of our sample theorem. With the goal represented as in 5.3 we obtain the following initial task.

$$\begin{array}{c} \left[\begin{array}{ll} (1) & (M \subseteq N \Leftrightarrow z \in M \Rightarrow z \in N)^- \\ (2) & (X \in 2^M \Leftrightarrow X \subseteq M)^- \end{array}\right] \\ \\ (A \subseteq B \Rightarrow 2^A \subseteq 2^B)^+ \end{array} \tag{5.4}$$

How do we begin our proof ? As a first step, we might want to proof $2^A \subseteq 2^B$ under the additional hypothesis $(A \subseteq B)^-$. We can easily perform this step by applying α-decomposition to the goal (this is achieved by clicking on the *Decompose Goal* link in Fig. F 23). The task we obtain is the expected

$$\begin{array}{c} \left[\begin{array}{ll} (1) & (M \subseteq N \Leftrightarrow z \in M \Rightarrow z \in N)^- \\ (2) & (X \in 2^M \Leftrightarrow X \subseteq M)^- \\ (3) & (A \subseteq B)^- \end{array}\right] \\ \\ (2^A \subseteq 2^B)^+ \end{array} \tag{5.5}$$

where $(A \subseteq B)^+$ occurs as a new support window. We can now apply the definition of $\subseteq$ (5.5.1) to the goal formula $(2^A \subseteq 2^B)^+$. In the GUI this is done by clicking on the corresponding definition. CORE then uses a heuristic (cf. Sec. 5.1) to compute those replacement rules from the selected formula that are most likely to be the appropriate rules to apply this formula. From the suggested rules the user then has to select the correct rule and apply it to the goal. In the current situation the heuristic computes the single applicable rule (i.e. $(M \subseteq N)^+ \rightarrow < (z \in M \Rightarrow z \in N)^+ >$) which is then applied automatically to the correct term position. One interaction has taken us to the new task

$$\begin{array}{c} \left[\begin{array}{ll} (1) & (M' \subseteq N' \Leftrightarrow y \in M' \Rightarrow y \in N')^- \\ (2) & (X \in 2^M \Leftrightarrow X \subseteq M)^- \\ (3) & (A \subseteq B)^- \end{array}\right] \\ \\ (z \in 2^A \Rightarrow z \in 2^B)^+ \end{array} \tag{5.6}$$

As a result of applying the replacement rule we have obtained the new goal $(z \in 2^A \Rightarrow z \in 2^B)^+$. Note that although the application of the replacement rule has instantiated the variables M and N in (5.6.1), we now have an uninstantiated version with fresh variable copies M' and N' in the supports. (Technically the system has increased the multiplicity of the equivalence (5.6.1) and a window for the newly created copy has been added to the task, while the window on the instantiated version is removed from the task). From now on we always assume that after application of an assertion we have an uninstantiated version in the supports, without indicating this explicitely.

In a next step, we apply the definition of powerset (5.6.2) to the formulas $z \in 2^A$ and $z \in 2^B$ respectively. We do this as before by selecting the corresponding assertion (5.6.2). In fact we have to repeat the relevant step for each of the two subformulas. Note that after clicking on the assertion (5.6.2) the system suggests to apply the rule $X \in 2^M \rightarrow< X \subseteq M >$ where we merely have to determine the application position, i.e. $(z \subseteq A)$ or $(z \subseteq B)$. When we apply 5.6.2 again we are now presented with two suggestions as how to apply 5.6.2; these are $X \in 2^M \rightarrow< X \subseteq M >$ and $X \subseteq M \rightarrow< X \in 2^M >$. When selecting the former rule the system automatically applies the rule to the unique application condition. We obtain the new task

$$\left[\begin{array}{ll} (1) & (M' \subseteq N' \Leftrightarrow y \in M' \Rightarrow y \in N')^- \\ (2) & (X \in 2^M \Leftrightarrow X \subseteq M)^- \\ (3) & (A \subseteq B)^- \end{array}\right] \qquad (5.7)$$

$$(z \subseteq A \Rightarrow z \subseteq B)^+$$

A further α-decomposition results in

$$\left[\begin{array}{ll} (1) & (M' \subseteq N' \Leftrightarrow y \in M' \Rightarrow y \in N')^- \\ (2) & (X \in 2^M \Leftrightarrow X \subseteq M)^- \\ (3) & (A \subseteq B)^- \\ (4) & (z \subseteq A)^- \end{array}\right] \qquad (5.8)$$

$$(z \subseteq B)^+$$

How to proceed ? At this stage, a human mathematician would probably argue that $(z \subseteq B)^+$ can be inferred from (5.8.3) and (5.8.4) with the help of a lemma which states that $\subseteq$ is a transitive relation, which we don't have at our disposal. Later we will see how the system can be further developed to suggest to close this goal by application of a method that corresponds to application of just this lemma. However, currently we are left with no other choice as to further expand definitions, before we can close the goal. That is, we basically have to proof the lemma by hand.

We want to apply definition (5.8.1) to (5.8.3), (5.8.4) and the goal. For the goal this is easy. We simply select (5.8.1) for application. The system heuristically computes a single correct replacement rule to apply (5.8.1) and hence applies it automatically. Further application of (5.8.1) to (5.8.3) and (5.8.4) can be achieved by first making (5.8.3) and then (5.8.4) the goal formula. On the GUI we do this by using the link *Make goal* that is attached to every non-axiom (cf. Figure F 23). Internally this link applies the *shift-rule*. The task after expansion of $(z \subseteq A)^-$ and $(A \subseteq B)^-$ is

$$\begin{array}{c}\left[\begin{array}{lc} (1) & (M' \subseteq N' \Leftrightarrow y \in M' \Rightarrow y \in N')^- \\ (2) & (X \in 2^M \Leftrightarrow X \subseteq M)^- \\ (3) & (A \subseteq B)^- \\ (4) & (z \subseteq A)^- \\ (5) & (y \in z \Rightarrow y \in B)^+ \\ (6) & (y \in z \Rightarrow y \in A)^- \end{array}\right] \\ \\ (y \in A \Rightarrow y \in B)^- \end{array} \tag{5.9}$$

Where (5.9.5) and (5.9.6) originates from the expansion of $(z \subseteq B)^+$ and $(z \subseteq A)^-$. The new goal is obtained through application of (5.8.1) to (5.8.3). After making (5.9.5) the goal window and a further α decomposition to it we obtain

$$\begin{array}{c}\left[\begin{array}{lc} (1) & (M' \subseteq N' \Leftrightarrow y \in M' \Rightarrow y \in N')^- \\ (2) & (X \in 2^M \Leftrightarrow X \subseteq M)^- \\ (3) & (A \subseteq B)^- \\ (4) & (z \subseteq A)^- \\ (5) & (y \in z)^- \\ (6) & (y \in z \Rightarrow y \in A)^- \\ (7) & (y \in A \Rightarrow y \in B)^- \end{array}\right] \\ \\ (y \in B)^+ \end{array} \tag{5.10}$$

This goal can now be closed by application of (5.10.7), (5.10.6) and finally (5.10.5) to the goal. During all this steps the system computes the correct suggestions as to which replacement rule realizes the application of the respective hypothesis so that the user is only concerned with selecting the appropriate support window for application. The remaining proof steps are then

$$\begin{array}{ccccccc} \left[\begin{array}{c} (1) \\ \vdots \\ (13) \end{array}\right] & \leadsto & \left[\begin{array}{c} (1) \\ \vdots \\ (13) \end{array}\right] & \leadsto & \left[\begin{array}{c} (1) \\ \vdots \\ (13) \end{array}\right] & \leadsto & \left[\begin{array}{c} (1) \\ \vdots \\ (13) \end{array}\right] \\ (y \in B)^+ & & (y \in A)^+ & & (y \in z)^+ & & \mathit{true}^+ \end{array} \tag{5.11}$$

We have obtained an agenda with a single task which can be closed and removed from the agenda by application of the *close* rule. This concludes our interactive proof of the theorem.

What does this example illustrate? We can identify three basic advantages that arise when we make use of the task structure.

1. The main feature of the task-structure is that although it allows to conduct proofs in a way that resembles ND style proofs, it hides the calculus level from the user. Notice that instead of applying different ND caclulus rules we can now apply formulas to the goal window in a uniform way.

2. Although the proof steps still resemble the essential steps in an ND proof, proofs conducted with the task structure are shorter. For instance, we

do not need to apply quantifier elimination rules because this is done implicitly by CORE.

3. Tasks allow us to present assertions in an intuitive way to the user. This is a prerequisite for making use of the facilitated way to apply these assertions. Together with the fact that we avoid a branching of the proof when we apply assertions (as would for instance be the case when we apply the $\Rightarrow_E$ rule backwards in ND proofs) this provides a convenient way to construct proofs.

Overall, we can see that the proof above does not require any logical knowledge about a certain calculus. Rather the tasks structure enables the user to concentrate on the essential steps of selecting and applying formulas contained in the supports of a goal. Although the application of an assertion still requires to choose between a number of replacement rules we have already significantly reduced the number of possible options by concentrating the computation of replacement rules to a particular assertion. In the next section we will see how this choice can be further reduced by a simple heuristic.

5.1 Computing Replacement Rules

Throughout the chapter we have often mentioned a heuristic that, given a support window and a goal window, computed those replacement rules that are most likely to be appropriate for application of the knowledge contained in the content of the support window to the goal. We will now examine this heuristic in more detail. Let us therefore note that the formulas in the goal and support window each correspond to a subtree of the IFT for the current problem. We will refer to the root node of the support formula as *assertion occurrence* and to the root node of the goal formula as *goal occurrence*. The question is then: how can we apply the label of the assertion occurrence to the label of the goal occurrence? We begin with making two observations

1. We can only apply a replacement rule $\mathcal{R}$ to a formula F if the input i of $\mathcal{R}$ unifies with a subterm of F.

2. Two formulas can only unify if their head-symbols are equal.

The heuristic that we will develop here will makes use of these facts and tries to find the minimal occurrence below the assertion occurrence that potentially unifies with an occurrence below the goal occurrence. To find such candidates we compare the head symbol of putative input occurrences with the head symbols of occurrences below the goal occurrence. This occurrence, if it exists, will then be used as an input for the replacement rule we generate from the assertion. Note that this input occurrence uniquely determines a replacement rule if we require that only literals occur as values in the rule. In case there are multiple occurrences $o_1, \ldots, o_n$ that qualify as input occurrence we suggest one replacement rule for each $o_i \in min\{o_1, \ldots, o_n\}$[2]. The reason for using minimal

[2]There can be multiple minima, because $\prec$ is a partial ordering

occurrences is that if $o \prec o'$ in an IFT then o is likely to have less β-related occurrences than o'. Accordingly, replacement rules with minimal input occurrences will tend to have less value occurrences and therefore introduce fewer subgoals. We therefore always suggest those rules for application that introduce the least number of new subgoals. Note that in order to maintain completeness the user can always request to see *all* replacement rules that can be obtained from the particular support window.

$$\begin{tikzcd}[column sep=small]
 & \subseteq(M^\gamma,N^\gamma) \Leftrightarrow \in(x^\gamma,M^\gamma) \Rightarrow \in(x^\gamma,N^\gamma)^0 \arrow[dl, dash] \arrow[dr, dash] & & & \wedge(\subseteq(\cup(X,Y),Z^\delta),F)^+ \arrow[dl, dash] \arrow[dr, dash] & \\
\boxed{\subseteq(M,N)} & & \in(x^\gamma,M^\gamma) \Rightarrow \in(x^\gamma,N^\gamma)^0 & \boxed{\subseteq(\cup(X,Y),Z^\delta)^+} \arrow[dl, dash] \arrow[dr, dash] & & F^+ \\
 & & \cup(X,Y)^+ & & Z^{\delta+} &
\end{tikzcd}$$

(5.12)

Figure F 13: Assertion (left) and goal occurrence (right) for the example mentioned above. M^γ and N^γ are freely instantiable variables, Z^δ is a δ type variable and $F \in \mathcal{L}$.

Before we formally present the algorithm, let us first look at an example.We take the assertion occurrence to be labeled with the already familiar formula $M^\gamma \subseteq N^\gamma \Leftrightarrow x^\gamma \in M^\gamma \Rightarrow x^\gamma \in N^\gamma$. We assume further that the goal occurrence is labeled with a formula $(X \cup Y \subseteq Z^\delta) \wedge F$. The IFTs for these formulas are shown in Fig. F 13 (we use prefix notation to make it easier to identify the head symbols). We can see that the node labeled with $\subseteq (M^\gamma, N^\gamma)$ in the left tree is the minimal occurrence which qualifies as an input for a replacement rule according to the criteria mentioned above; i.e. it has the same head symbol as the framed occurrence in figure F 13 (right). The replacement rule that is suggested by the above heuristic therefore is

$$(M^\gamma \subseteq N^\gamma)^0 \rightarrow < (\in (x^\gamma, M^\gamma) \Rightarrow \in (x^\gamma, N^\gamma))^0 >$$

which is exactly what we need to apply the assertion to the goal $(X \cup Y \subseteq Z^\delta) \wedge F$.

The algorithm that identifies the possible input occurrences is shown below. Note that it excludes occurrences with a head symbol in $\{\wedge, \vee, \Rightarrow, \neg, \Leftrightarrow\}$ from the set of input occurrences. The reason is that the heuristic tries to find the "characteristic" non-logical symbol of the assertion from which it tries to generate a replacement rule.

In the algorithm below the function SUBFORMSWITHSYM(s, o) returns a set of all occurrences below o that are labeled with a formula with head symbol s. In fact, in the current implementation it also returns the term position of each such occurrence within $label(o)$, which allows to automatically apply the preferred rule if there is a unique application position.

The occurrences that are returned by this algorithm serve as input occurrences of the replacement rules which realize the application of the assertion.

Algorithm 5.1.1: COMPUTEINPUTOCC(*assertocc*, *goalocc*)

```
$s \leftarrow head(label(assertocc))$
$I \leftarrow SubformWithSym(s, label(goalocc))$
if $s \notin \{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\} \wedge I \neq \emptyset$
  then $min(I)$
  else $\begin{cases} \{c_1, \ldots, c_k\} \leftarrow childs(assertocc) \\ \bigcup_i ComputeInputOcc(c_i, goalocc) \end{cases}$
```

The heuristic can fail in two ways. In some cases, it might not return an appropriate replacement rule. To maintain completeness the user therefore always has the option to see all rules for a given assertion to select the correct rule by hand. Secondly, the suggested rule might not unify with a subformula of the goal. This is possible because we do not perform a full unification when we try to identify suitable input occurrences. The reason to avoid full unification however is that we are interested in fast, but shallow computations that do not keep the user waiting.

It is interesting to remark that we can phrase the problem of finding the appropriate input occurrence for a replacement rule also in more abstract terms in a way that resembles the *term indexing* problem (see [SRV01] for an overview).

More precisely, given a an assertion occurrence a and a goal occurrence o we are interested in finding the minimal childs $a_1, \ldots, a_n$ of a such that $R(a_i, o')$ holds for a binary relation R and a child o' of the goal occurrence. We have to choose R in a way that if $R(a_i, o')$ then any replacement rule with input a_i can be applied to o'. Ideally, we choose $R(a_i, o')$ to hold iff a_i and o' are unifiable and of different polarity. However, we have already said that we want to avoid unification. To increase the preciseness of our heuristic while maintaining efficiency we could extend the comparison of the head-symbols of a_i and o' to a comparison of all non-variable symbols in a_i and o (in [SRV01] this is also referred to as *outline indexing*). We would then have chosen R in a way that is more precise than the currently implemented approach, but still not as computationally expensive as a test that involves full unification.

Note that we do not actually index terms to decrease the search process, but only frame our problem in terms of term indexing. This allows for a more precise description and also sets the ground for employment of real term-indexing techniques in future work. However, note that we would also have to address the additional complication constituted by the fact that CORE is a higher-order framework.

In chapter 8 we show how the ΩANTS suggestion mechanism checks for the applicability of replacement rules by using full unification. To make this possible ΩANTS will perform these computations as parallel processes in the background in order not to postpone the user interactions.

5.2 Summary

In this chapter we have demonstrated the potential that the use of tasks can bring to interactive proof construction. In particular we have shown that the task datastructure eases the representation of subgoals together with its context and facilitates the application of assertions. Furthermore we have seen that tasks provide a way to hide admissible but irrelevant replacement rules from the user. We have also pointed out some open problems.

When stepping through the proof of theorem 5.0.1 we arrived at a stage where we saw the need to be able to apply larger proof steps (step 5.8). In the next chapter we will formally develop the notion of a proof-method for the CORE system. Methods will provide a way to encode abstract proof steps, such as lemma application or certain domain-specific proof technique for the use in CORE. We will define methods to operate on the task structure.

Chapter 6

Interactive Proof Planning in CORE

So far we have seen that proof search in CORE is realized through the application of replacement rules, which are generated from the context of the current focus. However, although we have noted that COREs reasoning style facilitates the application of assertions during a proof, the level of reasoning that is constituted by the application of replacement rules is often still more detailed than the level at which human mathematicians construct and communicate proofs. In fact, when looking at proofs in mathematical textbooks, we find that humans often justify a proof step by application of a certain reasoning pattern, such as the diagonalization principle, the induction argument or application of a particular lemma.

To be able to support a more user friendly style of interactive proof construction, the concept of a *tactic* was introduced to interactive theorem proving [GMW79, Mil85]. Traditionally, tactics are understood as programs that execute a sequence of frequently occurring inference steps automatically. Accordingly, application of a tactic transforms an open goal into a (possibly empty) list of subgoals. Furthermore, tactics and inference rules can usually be combined to more complex tactics by so called *tacticals* such as *Repeat, If, Else*, etc. This enables a hierarchical structuring of tactics and the encoding of general proof strategies. This notion of a tactic is now used in many interactive theorem provers, such as ISABELLE [Pau94], NUPRL [CAB+86] and others, where the available tactics range from simple ones which, for example, only apply an inference rule for quantifier elimination repeatedly, to tactics that perform a full fledged proof-search automatically (e.g. the *Blast* tactic of the ISABELLE system employs a full matrix search).

Motivated by the insight that humans plan their proofs at various levels of abstraction, but not on calculus level, Bundy [Bun88] suggested to augment tactics by pre- and postconditions to be able to use them as planning operators (*methods*). This makes it possible to plan proofs automatically at the abstract level represented by the pre- and postconditions of the methods. The plan returned by a planning algorithm can then be refined (expanded) to calculus level by application of the tactics that are associated with each of the methods that occur in the plan.

In this chapter we develop the concept of a method for the CORE system. To do so, we first compare the notions of tactics and methods that are used in the ΩMEGA system. We will find that the concepts of tactics and methods in ΩMEGA are not much different. Based on this insight we set out to devise a concept of a method that can be used for interactive proof development as well as for proof planning, thus rendering it unnecessary to have different modules and datastructures for tactical theorem proving and proof planning. This will make it easier to work on the proof interactively as well as with the help of an automated proof planner. Furthermore, the declarative specification of a method will enable us to check for applicability of methods with a suggestion mechanism as we will see in Chapter 7. However, in this thesis, we will not be concerned with devising a planning algorithm for this methods. Rather we assume that we will use methods to perform abstract proof steps interactively.[1]

In Chapter 5 we saw the need for the application of the following lemma

$$\forall A_{Set}, B_{Set}, C_{Set}. A \subseteq B \wedge B \subseteq C \Rightarrow A \subseteq C \tag{6.1}$$

which we will use as an example to illustrate the concepts we introduce in this chapter.

6.1 Tactics and Methods in ΩMEGA

In ΩMEGA, tactics are the objects that represent abstract reasoning steps in interactive proof development, while methods are the corresponding datastructures for automated proof-planning.

However, in contrast to the traditional understanding of a tactic described above, tactics in ΩMEGA not only consist of a program. Instead, tactics are represented by a declarative and a procedural part. The declarative content specifies sets of premises and conclusions that have to be matched with proof lines of ΩMEGAs proof data structure $\mathcal{PDS}$ when a tactic is applied. This declarative part of a tactic thus defines an abstract inference step. The procedural part of a tactic is essentially an *expansion function* that is used to translate the abstract proof step described by the tactic into a sequence of proof steps in ΩMEGAs natural deduction calculus. This expansion function has to be executed at some point during the proof to show that the abstract proof step performed by the tactic was actually correct. The expansion function can again make use of other tactics which allows for the use of multiple abstraction levels.

The declarative part of a tactic in ΩMEGA can be graphically depicted as

$$\frac{P_1 \quad \dots \quad P_n}{C}\ \mathcal{T}\ (\pi_1, \dots \pi_j)$$

[1]This is what we call Interactive Proof Planning

where $\mathcal{T}$ is the name of the inference step. $\{P_1, \ldots, P_n\}$ are the premises and C is the conclusion of the tactic.[2]. Along with a set of additional parameters $\{\pi_1, \ldots, \pi_j\}$ these parameters are called the *formal arguments* of a tactic. Premises and conclusions of a tactic are described by formula schemes that constrain possible instantiations for this arguments. Furthermore, the declarative part can also contain a set of arbitrary *sideconditions*[3] that must hold for the tactic to be applicable.

Application of a tactic proceeds in essentially two steps: in the first step the formal arguments of a tactic have to be matched with proof lines of ΩMEGAs proof data structure (in case of premises and conclusions). Additional arguments can be instantiated with appropriate parameters, such as terms, substitutions or term-positions. The mapping of formal arguments to *actual arguments* of a tactic is referred to as a *partial arguments instantiation* (PAI) or *tactic (method) matching*. Each tactic matching determines the effect of the particular instantiation of a tactic on the proof state in the sense that each combination of instantiated and non-instantiated arguments corresponds to a different application direction of the tactic. We will speak of *forward application* when some of the premises, but not the conclusion of a tactic are instantiated with actual arguments (i.e. proof lines in the case of ΩMEGA). *Backward application* refers to a situation where the conclusion but not all premises of a tactic are instantiated [4].

To see in more detail what the putative instantiations of the premises and conclusion of a tactic are, we need to take a closer look at ΩMEGAs proof datastructure $\mathcal{PDS}$ [CS00]. The main entity of this datastructure are so called *proof lines* (see Chapter 2). We distinguish between open lines that represent open subgoals and closed lines. Tactics and methods use extra datastructures (foci and tasks respectively) which relate each open line to a subset of the closed lines that can be used to derive the respective open line. One also speaks of the closed lines that are related to an open line as *support lines* for an open line.

When a tactic is applied, the conclusion of the tactic has to be instantiated with an open line, while instantiations of the premises have to come from the corresponding support lines for that goal. Note that proof lines can only serve as instantiation for an argument if their associated formula matches the formula scheme of the respective argument.

The effect of a partial argument instantiation for a tactic on the proof state is the following: non-instantiated premises are introduced to the $\mathcal{PDS}$ as new open lines which (generally) inherit the supports from the instantiation of the conclusion. Non-instantiated conclusions become new support nodes, while instantiations of conclusions are removed from the list of open lines as they represent closed subgoals.

Methods in ΩMEGA (cf. [Mel98, EM99]) are represented in frame like datas-

[2]In ΩMEGA it is possible to specify tactics with multiple conclusions, however for the sake of simplicity we will concentrate on cases with only one conclusion.

[3]Sideconditions are also referred to as *application conditions*.

[4]Note that this definition subsumes the frequently used notion of sidewards application if we assume that tactics have only one conclusion.

tructures similar to the original suggestion by [Bun88]. They show the same distinction between a declarative and procedural part that we find for ΩMEGA tactics.

The declarative content is represented by slots for premises, conclusions, application conditions and declarations. Of main concern are the slots for premises and conclusions which together provide a logical specification of the inference step represented by the method. Premises and conclusions can be annotated by $\oplus$ and $\ominus$ to specify the effect of the method on the planning state. Here $\ominus$ means that a premise P labeled with $\ominus$ must be instantiated with a support line which will be removed from the support lines of the respective goal when the method is applied. A $\oplus$ premise remains uninstantiated and is introduced as new open line. Similarly, conclusions annotated with $\ominus$ are removed as open lines while $\oplus$ conclusions will be added as supports.

The procedural content of a method essentially consists of a proof scheme (i.e. a natural deduction proof segment where the justification of each line can again be a method) that can be seen as an instruction of how to expand the abstract reasoning step to less abstract inference steps and eventually to calculus level. This again makes it possible to structure methods hierarchically by nesting methods in the procedural part.

Application conditions of a method are decidable predicates that are evaluated when a method is applied; i.e. a method is only applicable in a given proof state when its application conditions all evaluate to true. The purpose of application conditions is to enable the user to encode heuristic knowledge as to when apply a method, into the planning process. As an example, take the `PeanoInduction` method [EM99] of ΩMEGA. It tries to close a line $\forall n.P(n)$ by application of the induction axiom for natural numbers. However, this concrete instance of the induction principle requires n to be a natural number. The side-condition of the method therefore checks whether n is indeed a natural number, before the method is applied.

As can be seen, tactics and methods in ΩMEGA differ in other aspects than traditional tactics and methods. To describe this difference, we have to distinguish between an inference step and its application directions. An inference step always states that certain conclusions can be derived from particular premises. However, each inference step can in general be applied in different directions. In a very broad sense we can therefore say that ΩMEGA tactics specify an inference step while ΩMEGA-methods define an application direction of an inference step (where the application direction is determined by the $\ominus$, $\oplus$ annotations).

6.2 Methods in CORE

When we now define methods for the CORE system we want to separate the abstract inference step from the computations necessary to apply this inference step in a particular direction. In the following we therefore introduce two objects: *methods* and *application directions.* Methods will consist of a declarative description of an inference step, together with the information common to all application directions of that inference step (e.g. the expansion function and the formula schemes of the arguments). Application directions describe computations necessary to apply a method in a particular direction.

The reason for this decision is twofold. Firstly, we note that an inference step with n arguments can theoretically be applied in $2^n - 1$ different directions. If we had to define a different method for each such application direction this would constitute a serious problem to a user who wants to select a method for application interactively. In ΩMEGA this problem can be ignored because methods are generally only used in automated proof planning. Furthermore, many abstract inference steps used by ΩMEGAs planner MULTI are only applied in particular directions anyway. However, the approach that we develop here will be suited for interactive method application too, because the user only needs to get in contact with the name and abstract description of the inference step (i.e. the method), but not with the application directions.

Secondly, moving the information that is shared between all application directions of an inference step to a single object (i.e. a method) removes some redundancy in the declaration of an inference step, because this information only has to be specified once. This is for instance the case for the formula schemes of a methods arguments. Because these formula schemes are identical for each application direction, we want to state them only once for all application directions.

Before we now formally define methods and application directions for the CORE system let us note that the distinction between inference steps and application direction is not entirely new, but in principle reflects the way tactics are implemented and represented in ΩMEGA.

We now define methods as descriptions of abstract inference steps that can be applied in different directions via so called *application directions.*

Definition 6.2.1 *(Method) A* method *is a tuple $\mathcal{M} = (P, C, A, o, FS, exp_{\mathcal{M}})$ where P (*premises*) and C (*conclusions*) are lists of argument names and C is non-empty. A is a possibly empty list of additional parameters. FS is a set consisting of one formula scheme for each argument name in $P \cup C$. $exp_{\mathcal{M}}$ is the* expansion function *of the method. Finally, o is an partial function (called an outline mapping) that maps an $K \in 2^{P \cup C}$ to an application direction of $\mathcal{M}$.*

Sideconditions that specify when a method is applicable are part of the application directions that are introduced below.

We can represent a method $\mathcal{M}$ with argument names $P = \{p_1, \ldots, p_n\}, C = \{c_1, \ldots, c_k\}$ additional parameters $T = \{\pi_1, \ldots, \pi_l\}$ and formula schemes $FS = \{P_1, \ldots, P_n, C_1, \ldots, C_k\}$ graphically as

$$\frac{p_1 : P_1 \quad \ldots \quad p_n : P_n}{c_1 : C_1 \quad \ldots \quad c_k : C_k} \mathcal{M}(\pi_1, \ldots, \pi_l)$$

For a concrete example let us consider a trivial method that encodes the sample lemma 6.2 as a proof method and can hence be used to apply the transitivity of $\subseteq$ in a proof.

$$\frac{p_1 : A \subseteq B \quad p_2 : B \subseteq C}{c : A \subseteq C} \subseteq -TRANS \tag{6.2}$$

where A, B, C are meta-variables that range over formulas. $A \subseteq B$ is therefore a formula scheme that specifies structural requirements for possible instantiations of the argument p_1. In Section 6.8 we describe how these formula schemes can be declaratively stated in a method specification. However, currently formula schemes are still expressed as *Lisp*-predicates that test whether a putative instantiation is of a certain syntactical form. Note that the application of the $\subseteq$-transitivity lemma is a simple inference step that would not usually be statically encoded as a method. Rather, one would employ this lemma as an axiom. In general, lemma-application should only be encoded as a method if the application of the lemma requires some additional (heuristic- or procedural-) knowledge that needs to be encoded in the sideconditions or computation functions of a method. An example is the application of the induction principle which often requires knowledge as to how to find a well-founded ordering on a given set. However, we will use method 6.2 to illustrate the basic concepts introduced in this chapter.

Having introduced the concept of a method which enables us to specify abstract inference steps we still need to make precise the notion of an application direction that represents the knowledge about how to apply an inference step in a particular direction.

Method application directions realize the different application directions of a method $\mathcal{M}$ in the following way: assume that $K \subseteq (P \cup C)$ is the set containing the arguments of $\mathcal{M}$ which become instantiated when a method is matched with a task. Then each such K corresponds to a different application direction of $\mathcal{M}$. The effect of applying a method in that direction is described by the method application direction $D_K^{\mathcal{M}}$ to which K is mapped by the outline mapping (i.e. $o(K) = D_K^{\mathcal{M}}$). Note that if $o(K)$ is undefined for a K then the respective method cannot be applied in that direction.

Definition 6.2.2 *(Method Application Direction) A* method application direction *$D_K^{\mathcal{M}}$ for a method $\mathcal{M} = (P, C, A, o, FS, exp_{\mathcal{M}})$ and a $K \subseteq P \cup C$ is a tuple $D_K^{\mathcal{M}} = (SC, COMP)$ where SC is a set of predicates (called* sideconditions*) and $COMP$ contains one* computation function *c_k for each k in K.*

The application direction $D_K^{\mathcal{M}}$ for a given K will first evaluate its sideconditions with respect to the instantiations $< K >$ of the arguments K. If the sideconditions evaluate to true the method is considered applicable. For each non-instantiated argument $k \in (P \cup C)\backslash K$ there has to be a computation c_k in the computations $COMP$ of the corresponding application direction. Each of this functions c_k takes as input the instantiations of the parameters K to compute the instantiation $< k >$ of argument k. Note that in the following we agree on the notation that if k is an argument of a method then $< k >$ refers to the instantiation of k or the result of the computation function for k.

After having applied all computation functions of the application direction, each argument of a method is instantiated with either a node from the FVIFT or a term computed from a computation function. We will then say that the method has a *completed outline*. This completed outline contains all information necessary to apply a method to a proof state.

We will later map the completed outline to a *state transformation rule* that realizes the effect of the method application on the CORE proof state. However, before considering this in more detail, it is necessary to think about what it means to instantiate arguments of a method in CORE.

6.2.1 Method matchings in CORE

We define the notion of a *method matching* [5] for the CORE system in a way similar to the method matchings (partial argument instantiations) in ΩMEGA. In the context of the ΩMEGA system methods operate on tasks $\Gamma \rhd \Delta$ which are basically sets of proof lines. Tasks in CORE consists of pointers (windows) to subtrees of the FVIFT. It is therefore obvious that, as a result of matching a method with a task in CORE, the arguments of the method should become instantiated with occurrences inside the windows of the task. The matching process itself is a pattern matching process between the formula schemes of the arguments and the labels of the occurrences in the windows of a task.

To see which occurrences qualify as valid instantiations for the arguments of a method notice that we can map a method

$$\frac{p_1 : P_1 \quad \ldots \quad p_n : P_n}{c : C}\ \mathcal{M}$$

back to an axiom scheme $(P_1 \wedge \ldots \wedge P_n \Rightarrow C)^-$ which is available in the context of every window G (hence the negative polarity). Annotation of this signed formula with uniform notation yields $(P_1^+ \wedge \ldots \wedge P_n^+ \Rightarrow^\beta C^-)^-$. From this annotated formula one can read of, that instantiations for the premises $P_1, \ldots, P_n$ can only be instantiated with occurrences which have a negative or an undefined polarity (such as equivalences or equations). The instantiation of the conclusion of a method has to be of positive or undefined polarity.

However, a closer look reveals that there are more constraints on possible instantiations for the arguments then just the polarity. Looking at putative instantiations for the premises we see that they must be independent. Intuitively this becomes clear when we consider a task $\Sigma, (A^+ \Rightarrow^\beta B^-) \rhd G$. In this situation B^- is *dependent* on A^+ that is, if we were to use B^- as an instantiation for the premise of a method the validity of the conclusion would suddenly depend on the validity of A^+.

In the following definition we therefore exclude dependent occurrences from the set of admissible instantiations for the arguments of a method. Later this additional constraint will turn out to be very helpful.

[5] In proof planning this is also referred to as an *action*.

Definition 6.2.3 *(Method Matching) Let $\mathcal{M} = (P, C, A, o, FS, exp_{\mathcal{M}})$ be a method and $T = \Sigma \rhd G$ a task for the current* FVIFT *R. Furthermore let R' be the smallest subtree of R that contains all windows in T. Then a* method matching *for $\mathcal{M}$ with respect to T is a mapping*

$$match_{\mathcal{M}} : P \cup C \to OCC_T$$

where OCC_T is the set of all occurrences inside the windows in task T and the following holds for each p and o with $match_{\mathcal{M}}(p) = o$:

1. *If $p \in P$ then o is independent in R' and $polarity(o) = -$.*
2. *If $p \in C$ then $polarity(o) = +$.*
3. *If $p \in C$ then o lies inside the goal window G.*
4. *If $p \in P$ then p lies in any of the windows in Σ.*
5. *$label(o)$ matches the formula scheme for argument p.*
6. *The application direction $o(K)$ for the set $K = \{p \in P \cup C | \exists o.match_{\mathcal{M}}(p) = o\}$ is defined.*

The third condition restricts instantiations of the conclusion of a tactic to occurrences inside the goal window. This is however no severe restriction. We introduce it only because it is convenient for interactive method application. For automated proof planning this additional condition can be ignored or modified.

Note that we will write $match_{\mathcal{M}}(p) = \epsilon$ for $p \in P \cup C$ if $match_{\mathcal{M}}(p)$ is not defined. It is furthermore worth pointing out that it is sufficient to require that occurrences in the range of a method matching are unconditional in R' in order to avoid the unwanted effects we mentioned earlier. Given a method $\mathcal{M}$ with arguments $P \cup C$ and $K \subseteq P \cup C$ we say that a method matching $match_{\mathcal{R}}$ describes the application direction $D_K^{\mathcal{M}} = o(K)$, determined by K iff $match_{\mathcal{M}}(k) \neq \epsilon$ for all $k \in K$.

Before we can apply a method matching $match_{\mathcal{M}}$ to a task we test whether the sideconditions of the corresponding application direction hold. If this is the case we carry out the computation functions of the application direction. By performing this computations we obtain the instantiations of those arguments that are not already instantiated by $match_{\mathcal{M}}$. We thus compute a completed outline for $\mathcal{M}$, which we defined formally as follows.

Definition 6.2.4 *(Completed Outline) Let $match_{\mathcal{M}}$ be a method matching for a method $\mathcal{M} = (P, C, A, o, exp_{\mathcal{M}})$ and $D_K^{\mathcal{M}}$ the application direction described by $match_{\mathcal{M}}$. The* completed outline *for $match_{\mathcal{M}}$ is then defined as a function*

$$outline^K_{match_{\mathcal{M}}} : P \cup C \to \mathcal{L}$$

where

$$outline^K_{match_{\mathcal{M}}}(k) = \begin{cases} label(match_{\mathcal{M}}(k)) & \text{if } match_{\mathcal{M}}(k) \neq \epsilon \\ c_k(k) & \text{otherwise} \end{cases}$$

and c_k is the computation function for $k \in D_K^{\mathcal{M}}$.

A completed outline for a method $\mathcal{M}$ contains all information that is necessary in order to apply a method matching to a proof state. Note that a completed outline maps each argument of a method to a formula in $\mathcal{L}$ and not to occurrences any more as does a method matching.

To be able to describe how a completed outline affects the task-structure and the underlying proof-state we have to distinguish between those formulas in the range of a completed outline that have been computed from the computation function of an application direction and those that are labels from argument instantiations. We therefore define two useful sets.

Definition 6.2.5 *Let $outline^K_{match_{\mathcal{M}}}$ be a completed outline for a method $\mathcal{M}$ with arguments K in $P \cup C$ then we define the sets $I_{\mathcal{M}}$ and $C_{\mathcal{M}}$ as follows*

- $I_{\mathcal{M}} = \{outline^K_{match_{\mathcal{M}}}(k) | k \in K\}$
- $C_{\mathcal{M}} = \{outline^K_{match_{\mathcal{M}}}(c) | c \in (P \cup C) \backslash K\}$

Intuitively, $I_{\mathcal{M}}$ is the set of all instantiations of arguments of $\mathcal{M}$ and $C_{\mathcal{M}}$ is the set of all values that are computed by the computation functions of the application direction $o(K)$. We generalize our notation $< . >$ for method matchings to completed outlines and refer to $outline^K_{match_{\mathcal{M}}}(a)$ as $< a >$ for an $a \in P \cup C$.

6.3 Proof State Transformation

Having defined what a method matching is and how to obtain a completed outline for it, we need to think about how such a matching transforms the proof state. We first describe the changes on an agenda that are caused by the application of a method matching to a task. Then we specify how we have to change the proof-state (i.e. the FVIFT) in order to be able to realize these changes.

Let us therefore assume that we want to apply a method matching $match_{\mathcal{M}}$ to a task $T = \Sigma \rhd G$ and that $match_{\mathcal{M}}$ describes a backward application of method $\mathcal{M}$ with premises $p_1, \ldots, p_n$ and conclusion c. To be able to apply $match_{\mathcal{M}}$ to T we first have to compute the completed outline $outline^K_{match_{\mathcal{M}}}$ for $match_{\mathcal{M}}$.

We denote with $p_{i_1}, \ldots p_{i_k}$ all those premises that are instantiated by $match_{\mathcal{M}}$ (i.e. $< p_{i_j} > \in I_{\mathcal{M}}$). Because we apply the method backwards we also have $< c > \in I_{\mathcal{M}}$. If $p_{u_1}, \ldots p_{u_l}$ are the arguments of $\mathcal{M}$ without an instantiation in $match_{\mathcal{M}}$ (i.e. $< p_{u_l} > \in C_{\mathcal{M}}$) then application of the method matching to T should result in the following effect on the task structure:

$$\Sigma \rhd G[< c >]_\pi \rightsquigarrow \begin{cases} \Sigma \rhd G[true^+]_\pi \\ \Sigma \rhd < p_{u_1} > \\ \vdots \\ \Sigma \rhd < p_{u_l} > \end{cases} \tag{6.3}$$

This means that the task $\Sigma \rhd G[< c >]_\pi$ is replaced on the agenda by a task where the instantiation of the conclusion $< c >$ is replaced by $true^+$ in the goal window (i.e. $\Sigma \ \rhd G[true^+]_\pi$). Furthermore each of the uninstantiated arguments p_{u_i} results in a new task with $< p_{u_i} >$ as goal window.

To see how forward application affects a task let the uninstantiated arguments in $match_{\mathcal{M}}$ be again denoted by p_{u_i}. Let furthermore $< c >$ be the instantiation of the conclusion (remember that we obtained this instantiation from an computation function because we are dealing with forward application). Then forward application of the method should affect the current task as follows [6]

$$\Sigma \rhd G \leadsto \begin{cases} \Sigma \rhd < p_{u_1} > \\ \vdots \\ \Sigma \rhd < p_{u_l} > \\ \Sigma, < c > \ \rhd G \end{cases} \tag{6.4}$$

Here we have added $< c >$ to the supports of our task, while a new task was created for each of the $< p_{u_i} >$.

In order to be able to perform these changes of the task structure we have to change the CORE proof state (i.e. the FVIFT) accordingly. Considering backwards application we have to replace $< c >$ in the FVIFT with $< p_{u_1} >, \ldots, < p_{u_l} >$ which have to be β-related between each other. By opening windows on each of the $< p_{u_i} >$ we can then make them the goal windows of new tasks as in 6.3. Note that the necessary transformation of the FVIFT could in principle be obtained by application of a replacement rule

$$< c > \rightarrow < p_{u_1} >, \ldots, < p_{u_l} > \tag{6.5}$$

to $< c >$.

For forward application the situation is not much different. We also want to introduce $< p_{u_1} >, \ldots, < p_{u_l} >$ as goal windows of new tasks (cf. 6.4) in order to treat them as new subgoals. However, we also need to add $< c >$ to the support windows of the current task. This means we have to introduce $< c >$ in α-relation to the goal window of the current task while $< p_{u_1} >, \ldots, < p_{u_l} >$ have to be β-related to $< c >$. This effect could again be realized with the help of a replacement rule.

$$< p_{i_l} > \rightarrow < c >, < p_{u_1} >, \ldots, < p_{u_l} > \tag{6.6}$$

Here $< p_i >$ is the instantiation of an arbritrary premise p_i. We need this instantiation only to introduce $< c >$ and the $< p_{u_1} >, \ldots, < p_{u_l} >$ into the context of G.

The problem we have with rules 6.6 and 6.5 is that they are in general not admissible in the context denoted by the current task. They represent speculative proof steps which we have to carry out by application of an oracle-rule. In the following, we will define how a completed outline for a given method matching can be mapped onto an oracle-rule. This oracle-rule we call a *State Transformation Rule.*

[6]Remember that we only distinguish between forward and backward application.

Definition 6.3.1 *(State Transformation Rule) If $\mathcal{M}$ is a method, $outline^{K}_{match_{\mathcal{M}}}$ a completed outline for this method and $I_{\mathcal{M}}$ and $C_{\mathcal{M}}$ as defined above, then the* state transformation rule $T_{match_{\mathcal{M}}}$ *for this outline is the following:*

$$T_{match_{\mathcal{M}}} = \begin{cases} [I_{\mathcal{M}} \backslash \{<c>\}] \ <c> \rightarrow C_{\mathcal{M}} & \textit{if} <c> \in I_{\mathcal{M}} \\ [I_{\mathcal{M}} \backslash \{<p>\}] \ <p> \rightarrow C_{\mathcal{M}} & \textit{else} \end{cases}$$

The rule for the case that $<c> \in I_{\mathcal{M}}$ realizes backward application of $\mathcal{M}$ while the rule for the other case corresponds to forward application. Note that in the latter case $<c> \in C_{\mathcal{M}}$, that is $<c>$ occurs on the right-hand side of the oracle rule as required in 6.6. Furthermore $<p>$ in $I_{\mathcal{M}}$ is an arbitrary element from $I_{\mathcal{M}}$ which is non-empty because we must have $I_{\mathcal{M}} \neq \emptyset$ in order for a method to be applicable. The transformation rule defined here only works for methods with one conclusion but can easily be generalized to methods with multiple conclusions.

The state transformation rules defined above differ from the rules 6.5 and 6.6 in that the labels of the instantiated arguments $I_{\mathcal{M}}$ appear as conditions in the state transformation rule while they did not occur in rules 6.5 and 6.6. The reason for this is simply that only terms that occur in an oracle rule can appear in the proof obligation which is introduced to warrant application of the oracle rule (cf. Section 3.7). Hence, without these parameters, the new proof obligation that results from the oracle application would be $(<p_{u_1}>, \ldots, <p_{u_l}> \Rightarrow <c>)^{+}$, which can in general not be proven.

The separation of instantiated arguments and uninstantiated arguments into condition and value part of the oracle is only technically motivated as will soon become clear. In principle, any distribution of the premises over conditions and values would have the same effect on the FVIFT, as we saw in Section 3.7.

Let us consider the application of the state transformation rule for backward application in more detail. We consider a method $\mathcal{M}$ and a completed outline $outline^{K}_{match_{\mathcal{M}}}$. Again, we denote the elements in $I_{\mathcal{M}}$ as $<p_{i_1}>, \ldots, <p_{i_k}>$. Similarly we refer to elements from $C_{\mathcal{M}}$ as $<p_{u_1}>, \ldots, <p_{u_l}>$. When we now apply the corresponding state transformation rule, we basically perform a cut over

$$<p_{i_1}> \wedge \ldots \wedge <p_{i_k}> \wedge <p_{u_1}> \wedge \ldots \wedge <p_{u_l}>$$

We have seen in Section 3.7 that this leads to a replacement of the subtree for $<c>$ by two β-related trees η and R. Let us look at these subtrees in turn. Because the conclusion of a method can only be instantiated with a positive occurrence the first subtree is of the form depicted in Figure F 14 and encodes

$$R = \alpha(\beta(\underbrace{<p_{i_1}>^{+}, \ldots <p_{i_l}>^{+}}_{Conditions}, \underbrace{<p_{u_1}>^{+}, \ldots <p_{u_l}>^{+}}_{Values})^{+}, <c>^{+})$$

Figure F 14: Result of State Transformation Rule. Occurrences are labeled as conditions and values as indicated.

the replacement of the subgoal $< c >$ by new subgoals. In Figure F 14 we have indicated which of the subgoals are classified as conditions and which as values in the oracle application. If we recall that the conditions $< p_{i_a} >^+$ originate from instantiations $< p_{i_a} >^-$ of a methods arguments, we can see that we can use this instantiations $< p_{i_a} >^-$ to close subgoals in the condition list immediately. We merely need to generate a replacement rule $< p_{i_a} >^+ \rightarrow true^+$ from each of the instantiations $< p_{i_a} >^-$. With these rules we can then close the condition subgoals. The remaining, simplified subtree is then the following

$$\alpha(\beta(\underbrace{< p_{u_1} >^+, \ldots < p_{u_l} >^+}_{Values})^+, < c >^+)$$

This is exactly the replacement of $< c >$ that we wanted (cf. 6.5). By mapping a completed outline to an oracle rule we were thus able to replace a goal $< c >$ with new subgoals $< p_{u_1} >^+, \ldots < p_{u_l} >^+$.

Now it becomes clear why we distributed the instantiations of the arguments over the value and condition part of the oracle rule. When applying the oracle rule we obtain two lists. One contains a window on each of the values, the ***value-list***, while the other list contains windows on the conditions of the rule and is hence called the ***condition list***. This means that after application of a state transformation rule we can immediately close all windows in the condition list with the help of the replacement rules generated from the $< p_{i_a} >^-$. Then we merely need to make the windows in the value list the goal windows of new tasks to obtain exactly the transformation of the agenda that we described in Section 6.3.

The second subtree that is introduced by the oracle rule represents the proof obligation that results from the oracle application. The tree will be of the form

$$\eta^+ = \alpha(\alpha(< p_{i_1} >^-, \ldots < p_{i_l} >^-, < p_{u_1} >^-, \ldots < p_{u_l} >^-)^-, < c >^+)^+$$

We refer to the formula represented by this tree as η^+. Later, we will see how we can define an expansion function for a method that, applied to η^+, closes this goal.

To update the task structure after application of a state transformation rule as described above, we extend the inference rules for tasks by a rule for oracle application.

$$\frac{\Sigma \rhd I}{\Sigma, I \rhd \eta^+, \Sigma, I \rhd R_1, \ldots, \Sigma, I \rhd R_n} \; Oracle \; [\Phi] \; I \rightarrow V$$

where $\{R_1, \ldots, R_n\} = \Phi \cup V$. This rule is in principle similar to the *apply* rule for replacement rule application. The only difference is that the rule above generates an additional task $\Sigma, G' \rhd \eta^+$ which encodes the obligation to show the validity of the oracle-step.

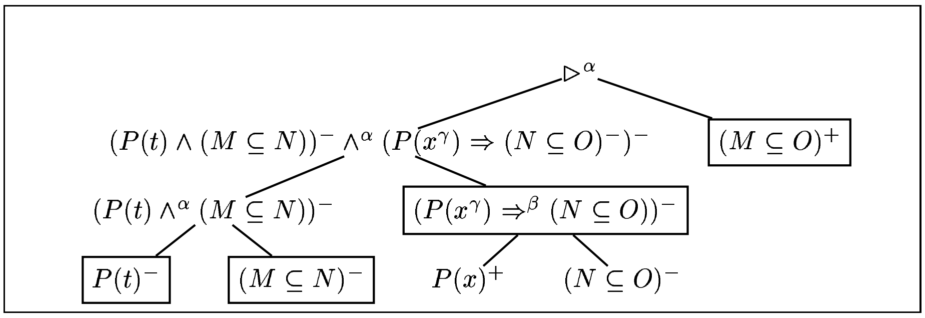
Note that we can use this rule also for backward application of a method. If we want to apply a method to a window $s \in \Sigma$ we simply use the *shift*-rule and make s the goal window, before we apply the state transformation with the help of the just defined rule.

6.4 Sample Method Application

To study how application of a method transforms the FVIFT and the task structure at hand of a concrete example let us again consider the method $\subseteq -TRANS$ (see 6.2) which applies the transitivity of the $\subseteq$-relation to the task

$$T = P(t)^-, (M \subseteq N)^-, (P(x^\gamma) \Rightarrow^\beta (N \subseteq O))^- \rhd (M \subseteq O)^+ \qquad (6.7)$$

A possible FVIFT that is represented by this task is shown in Figure F 15 below. Let us now try to apply the method backwards to this task. The first step will



$\rhd^\alpha$
$(P(t) \wedge (M \subseteq N))^- \wedge^\alpha (P(x^\gamma) \Rightarrow (N \subseteq O)^-)^-$
$(M \subseteq O)^+$
$(P(t) \wedge^\alpha (M \subseteq N))^-$
$(P(x^\gamma) \Rightarrow^\beta (N \subseteq O))^-$
$P(t)^-$
$(M \subseteq N)^-$
$P(x)^+$
$(N \subseteq O)^-$

Figure F 15: FVIFT for task 6.7

consist of the computation of a method matching for the method $\subseteq -TRANS$. What are the putative instantiations for the arguments of the method? By definition we have to restrict our search for instantiations of the conclusion to the positive occurrences in goal window of the task. Fortunately, we find that the only occurrence in this window $(N \subseteq O)^+$ matches the formula scheme $(A \subseteq C)$ for the conclusion. We then have to look for instantiations of the premises of the method within the support windows. We see that the occurrences $(M \subseteq N)^-$ and $(N \subseteq O)^-$ match the formula schemes for p_1 and p_2 respectively. Although both occurrences have a non-negative polarity, only $(M \subseteq N)^-$ qualifies as an instantiation of p_1. The reason is that that $(N \subseteq O)^-$ is dependent on $P(x^\gamma)$. Because $P(x^\gamma)$ occurs in the same minimal subtree that contains all windows of task T (i.e. the entire tree in Figure F 15) this is a violation of the requirement we made in the definition of a method matching and hence $(N \subseteq O)^-$ is no valid instantiation for p_2. We end up with the method matching

$$match_{\subseteq -TRANS} =< c : (M \subseteq O)^+, p_1 : (M \subseteq N)^- > \qquad (6.8)$$

From this method matching we obtain the following completed outline for $\mathcal{M}$

$$< c : (M \subseteq O), p_1 : (M \subseteq N), p_2 : (N \subseteq O) >$$

Note that we have $I_{\mathcal{M}} = \{(M \subseteq N), (M \subseteq O)\}$ and $C_{\mathcal{M}} = \{(N \subseteq O)\}$. This completed outline can then be mapped to a state transformation rule. According to definition 6.3.1 we obtain the state transformation rule

$$[(M \subseteq N)]\ (M \subseteq O) \rightarrow< (N \subseteq O) > \qquad (6.9)$$

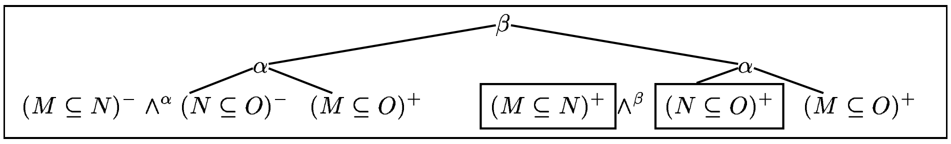

β
α
α
$(M \subseteq N)^- \wedge^\alpha (N \subseteq O)^-$
$(M \subseteq O)^+$
$\boxed{(M \subseteq N)^+} \wedge^\beta \boxed{(N \subseteq O)^+}$
$(M \subseteq O)^+$

Figure F 16: Replacement of $(M \subseteq O)^+$ in Figure F 15 as a result of method application.

which can then be applied to $< c >= (M \subseteq O)^+$. As a result, the occurrence $(M \subseteq O)^+$ in Figure F 15 is replaced by the tree in Figure F 16. We can see that the instantiation of the conclusion $(M \subseteq O)^+$ is now replaced by two β-related subtrees. The right subtree

$$R = \alpha(\beta\,\underbrace{((M \subseteq N)^+}_{Condition},\underbrace{(N \subseteq O)^+)}_{Values}, (M \subseteq O)^+)$$

contains new windows for $(M \subseteq N)^+$ and $(N \subseteq O)^+$ which represent alternative subgoals to the instantiation $< c >$ of the conclusion. The left subtree represents the new proof-obligation

$$\eta^+ := (M \subseteq N)^- \wedge^\alpha (N \subseteq O)^- \Rightarrow^\alpha (M \subseteq O)^+ \tag{6.10}$$

We first have a look on the windows for the new occurrences $(M \subseteq N)^+$ and $(N \subseteq O)^+$ in R. The window on the occurrence $(M \subseteq N)^+$ originates from the condition of the oracle-rule. Because every occurrence in the condition part corresponds to an actual argument of the method, we can use this instantiation to close the subgoal represented by this occurrence. In the example, the new occurrence originates from the instantiation $(M \subseteq N)^-$ of p_1 from which we can construct the rule: $(M \subseteq N)^+ \rightarrow true^+$ which proves the respective window.

Application of the tactic is completed when the rule $(M \subseteq N)^+ \rightarrow true^+$ is applied to the condition windows. We have then transformed the subtree R into

$$R' = \alpha(\beta(true^+, \underbrace{(N \subseteq O)^+}_{Values}), (M \subseteq O)^+)$$

It is worthwhile to point out that the intermediate step, which closes the condition window, is carried out automatically and is hence invisible to the user. The occurrence $(N \subseteq O)^+$ is thus the only real subgoal that is introduced through application of the method.

Eventually, we have to replace task 6.7 by tasks for subtrees R' and η on the agenda. This means that task 6.7 becomes replaced by

$$\left\{ \begin{array}{l} P(t), (M \subseteq N)^-, (\exists x.P(x) \Rightarrow (N \subseteq O)^-)^- \rhd (N \subseteq O)^+ \\ P(t), (M \subseteq N)^-, (\exists x.P(x) \Rightarrow (N \subseteq O)^-)^- \rhd \eta^+ \end{array} \right\}$$

As we pointed out above, the second task represents the obligation to show that the method application was indeed a valid step; i.e. we have to provide a proof of the lemma encoded by the method. Currently the system automatically closes this task by applying the method expansion function to it if this function is specified. Alternatively, one could delay the expansion of the method or even abandon it completely if one is not interested in a proof of the lemma.

6.5 Method Expansion

So far, we have defined what a method for the CORE system is and how we apply it to a given proof state. We have also seen that application of a method

$$\frac{p_1 : P_1 \quad \ldots \quad p_n : P_n}{c : C} \mathcal{M}$$

introduces a new proof-obligation

$$\eta^+ := (< p_1 > \wedge \ldots \wedge < p_n > \Rightarrow < c >)^+$$

with the corresponding task $T_{EXP} = \Sigma \triangleright \ (< p_1 > \wedge \ldots \wedge < p_n > \Rightarrow < c >)^+$ When we look at the above formula, it becomes clear that although the exact proof-obligation depends on the actual arguments $< p_i >$ and $< c >$, all proof-obligations that are the result of applying a particular method will usually share a common overall structure. Since in many cases methods are abstract proof steps that merely abbreviate a frequently used sequence of inferences, we can use this sequence of inference steps to close the proof-obligation η^+. This is what we will refer to as *method expansion.* In other words, for most methods the proof of the resulting proof-obligation η^+ follows a certain scheme which is known at the time the method is specified. We will encode this knowledge of how to expand a method (i.e. closing the resulting proof obligation) in little programs. We so obtain *expansion functions* that, applied to the proof-obligation, automatically apply CORE calculus rules to rewrite the proof-obligation to $true^+$.

As a simple example let us consider the expansion function of the $\subseteq -TRANS$ method. We have seen that after application of this method the resulting proof-obligation will be of the following form

$$(M \subseteq N)^- \wedge^\alpha (N \subseteq O)^- \Rightarrow^\alpha (M \subseteq O)^+$$

We can always obtain a proof of this goal by performing the following steps.

1. Search for the definition of $\subseteq$ [7] ; i.e $(A^\gamma \subseteq B^\gamma \Leftrightarrow x \in A^\gamma \Rightarrow x \in B^\gamma)^-$ in the support windows Σ of task T_{EXP}.

2. Generate the rule $(A^\gamma \subseteq B^\gamma)^0 \rightarrow < (x \in A^\gamma \Rightarrow x \in B^\gamma) >^0$ from this definition.

[7] We have encoded this definition as an assertion which is represented by some part of the FVIFT. However, although we know that it is contained in a support window of the current task, we do not know in which.

3. Apply this rule to occurrences $(M \subseteq N)^{-}, (N \subseteq O)^{-}$ and $(M \subseteq O)^{-}$ respectively which changes the content of the goal window to

$$(x \in M \Rightarrow x \in N)^{-} \wedge^{\alpha} (x \in N \Rightarrow x \in O)^{-} \Rightarrow^{\alpha} (x \in M \Rightarrow x \in O)^{+}$$

4. Focus to $(x \in O)^{+}$.

5. Generate and apply: $x \in O^{+} \rightarrow < x \in N^{+} >, x \in N^{+} \rightarrow < x \in M^{+} >, x \in M^{+} \rightarrow < \mathit{true}^{+} >$ in turn.

At the moment, we write expansion functions like the one above in COREs tactic specification language which is basically a *Lisp*-like language enriched by certain keywords (e.g. *try, fail,* ...) that facilitate backtracking. This means that the expansion functions we use are essentially CORE tactics.

6.6 Integration of Automated Reasoners

In ΩMEGA it is possible to integrate automated reasoning systems such as OTTER [McC90] and MACE [McC94] into the interactive proof process. To close subgoals automatically ΩMEGA provides tactics, such as

$$\frac{p_1 : P_1 \quad \ldots \quad p_n : P_n}{c : C}\ \textsc{Otter} \qquad\qquad \frac{p_1 : P_1 \quad \ldots \quad p_n : P_n}{c : C}\ \textsc{Mace}$$

that justify a goal C from the premises $P_1, \ldots, P_n$ by a call to the first-order theorem prover OTTER or the propositional logic decision procedure MACE. Tactics that incorporate other external reasoning systems, like TPS or computer algebra systems (CAS) have also been implemented in the system (see [BBS99] and [Sor01]). Proofs that are found by external theorem provers are translated to ΩMEGAs ND calculus in the expansion of the tactic.

Using the methods that were introduced in this chapter, we can integrate external reasoning systems into CORE in much the same way. If we extend the method specification language to allow for the encoding of methods with a flexible number of premises we can directly encode methods that carry out the inferences of the tactics above.

However, to realize expansion of these methods would require the adaptation of proof- and calculus transformation approaches like TRAMP (see [Mei00]) and SAPPER (see [KKS98] and [Sor00]) to CORE.

In Chapter 8 we will describe how the suggestion mechanism ΩANTS can be used to apply methods that make use of external reasoning provers to inactive tasks on the agenda. This will enable us to tackle inactive subproblems with automatic theorem provers in times when the user remains inactive.

6.7 Limitations of the Approach

The concept of a method that was introduced in this chapter roughly resembles the original concept of a method as suggested by [Bun88] where a method is a specification of an LCF style tactic. To see why, remember that we carry out the abstract inference step described by a method through application of a state transformation rule. This rule allows us to refine a goal to some subgoals. However, to justify this refinement, the system creates an additional proof obligation which we close by applying the expansion function of the method to it. Since we choose to implement this expansion function as an (LCF style) CORE tactic we can see a CORE method as a wrapper for a CORE tactic. In the long run one could envision a different expansion mechanism which makes it possible to describe the expansion of a method in a more declarative way.

However, the methods for the CORE system lack an important feature of the ΩMEGA methods. In CORE we can only use multiple levels of abstraction in the specification of a method if we compose the expansion tactic out of smaller tactics. In ΩMEGA we are able to use other methods in the expansion of a method. Also, while expansion of ΩMEGA methods can be declaratively specified in form of a partial proof fragment, the expansion mechanism we have introduced here restricts us to a procedural description of the expansion. What needs to be done at this point is to develop a declarative language for specifying expansions in which methods can be compounded to larger methods. This would basically mean to introduce *methodicals* (as defined by [RS01]) into our method specification language.

6.8 Syntax of the Specification Language

Now that we know what a method is in the context of the CORE system, we want to be able to specify such methods. We therefore introduce a language in which methods can be specified. This language is based on elements from the languages in which methods and tactics are specified in ΩMEGA. It aims at using declarative elements for the description of methods and application directions wherever possible to increase readability. A complete specification of the language for method declaration is given in form of a BNF-grammar in (Figure F 22, Appendix). [8]We will see that the distinction between methods and application directions is also reflected in the specification language. Let us first look at how we specify methods.

6.8.1 Specifying Inference Steps

A method (<inference>) is specified according to the following scheme.

<inference> ::= **(defmethod** <name>
    (in <theory>**)**
    (declarations
        (type-variables <typevar-list>**)**
        (variables <var-list>**)**

[8]The language described in this chapter already allows for the specification of multiple conclusions and hypothesis for particular premises which is not yet accounted for in our concept of a method.

```
        (methodvars    <mvar-list>))
      (parameters <name-list>)
      (premises    <arg-list>)
      (conclusions <arg-list>)
      (outline-mappings (<map-list>))
      (expansion <name>))
```

Here < name > is the name of the inference and < theory > is the theory to which the inference belongs. In the **declarations** part of a method we can introduce variables that will be visible in all parts of the method declaration as well as in the application directions that belong to the method. We distinguish between three kinds of variables. Type variables are variables that range over types. We can use any string (<name>) to declare a type variable:

```
<typevar-list>::=  <type-var> | ϵ
                   |<type-var> <typevar-list>
<type-var> ::=     <name>
```

In the **variables** section we introduce variables of a certain type according to the following grammar

```
<var-list> ::= <var> | ϵ
               | <var> <var-list>
<var>      ::= ( <name> <type> )
```

Types < type > can be defined according to the scheme below. Note that the < type-list > construct allows us to specify functional types. For instance the type list `(o N M)` describes the functional type $(M \to N) \to o$. Accordingly, `(a (o N M))` specifies a variable a of type $(M \to N) \to o$. Note that <basetype> comprises all types defined in theory <theory> and <typevar-list>.

```
<type> ::= <basetype>
           |<type-var>
           |(<type-list>)

<type-list>::= <type> | ϵ
               |<type> <type-list>
```

Method variables are variables of arbitrary type that can be used to store intermediate results computed by the computation functions in the application directions of a method. Types of method variables can be constructed out of simpler types with the help of type constructors such as

<constructor> ::= **listof** | **pairof** | ...

and the following grammar rules

```
<mvar-list> ::= <mvar> | ϵ
                | <mvar> <mvar-list>
<mvar> ::= ( <name> <mvar-type> )
<mvar-type> ::= <basetype>
                | ( <constructor> <mvar-type> )
```

Currently, **listof** and **pairof** are the only predefined type constructors. However, additional constructors and base-types can be introduced easily by the user of the system.

The then following parts of a method declaration introduce the arguments (premises and conclusions) as well as additional arguments (parameters) of a method. We can define any variable declared in the method to be a parameter by using the < name-list > construct. Premises and conclusions consist of arguments names together with a declarative description of the formula scheme <decl_content> for the respective argument.

```
<arg-list> ::= <arg> | ε
             | <arg> <arg-list>
<arg>      ::= ( <name> <decl_content> )
```

The declarative description < decl_content > can be any term over the the symbols and variables that are declared in the method or in the corresponding theory <theory>. As an example consider the already familiar formula scheme $\subseteq (A, B)$ which we would describe as `(subset A B)`, assuming that the symbol `subset` is defined in < theory > and `A` and `B` are variables of type `(o a)` for some type or type variable `a`.

By using higher-order variables in the declarations of formula-schemes, the method interpreter can compute values of non-instantiated arguments by a pattern-matching process. This pattern matching process subsumes the computation functions of the application directions of a method that were introduced in Section 6.2.

An important part of the method declaration is the specification of the outline mapping which is done according to the following part of the grammar [9] where < name > is the name of the application direction to which a method matching is mapped.

```
<map-list> ::= <mapping>
             | <mapping> <map-list>
<mapping>  ::= (({existing|nonexisting}*) <name> ) | ε
```

To map a method matching to an application direction we assume that the arguments of the methods are always ordered as follows, where $c_1, \ldots, c_k$ are the conclusions of the method, $p_1, \ldots, p_l$ are premises and $\pi_1, \ldots, \pi_n$ are the additional parameters:

$$O = [c_1, \ldots, c_k, p_1, \ldots, p_l, \pi_1, \ldots, \pi_n]$$

We then use a string $m = m_1 \ldots m_{|O|}$ over $\{existing|nonexisting\}^*$ of length $|O|$ to describe a set of method matchings as follows: m describes a matching $match_{\mathcal{M}}$ iff for all $i = 1, \ldots, |O|$ the following holds: $m_i = existent$ iff $match_{\mathcal{M}}(a)$ is defined for the argument a at position i in O.

This concludes our description of the specification of methods. We now investigate how application directions of a method are specified.

[9]Note the slight abuse of the notation, where we use a regular expression in the grammar.

6.8.2 Application Directions

Application directions (<appl-dir>) must be specified according to the following grammar

```
<appl-dir> ::= (defdirection <name> <name>
                 (declarations
                    (type-variables <name-list>)
                    (variables      <var-list>)
                    (methodvars     <mvar-list>))
                 (hypothesis <hyp-list> )
                 (premises <arg-list2> )
                 (conclusions <arg-list2> )
                 (application-conditions <list-of-expr> )
                 (outline-computations<comp-list>))
```

Here <name> and <name> correspond to the name of the application direction and the method to which it belongs, respectively. The declarations part is similar to the one used in <inference>, but all variables declared here are only locally visible to the application direction. In the hypothesis part

```
<hyp-list> ::= <hyp> | ϵ
             | <hyp> <hyp-list>
<hyp>      ::= ( <name> <decl-content> :for <name>)
```

we can introduce hypothesis for some of the methods arguments. In principle, the declaration of hypothesis is similar to the declarations of arguments in a method. We also specify a name and a formula scheme for the hypothesis. However, for each hypothesis that we introduce we have to give the <name> of the premise to which the hypothesis belongs. This is done with the help of the **:for** keyword above.

Next, we need to define for each argument of the method how an instantiation of this argument "behaves" in the task to which we apply the method in the respective direction; i.e. whether the instantiation of the argument should be added or removed from the task after application of the method. We do this by pairing each argument name with a sign $s \in \{+, -, *\}$:

```
<arg-list2> ::= <arg2> | ϵ
              | <arg2> <arg-list2>
<arg2>      ::= (+ <name> )
              | (− <name> )
              | (∗ <name> )
```

The semantics of this annotations is the following: arguments labeled with "+" are added as new windows to a task after application of the method. Accordingly, in application directions for backward application we can only label premises with "+" that do not become instantiated when applying the method. In application directions that realize forward applications, only conclusions can be labeled with "+".

Exactly the converse is indicated by a "−" label. Arguments with this label are removed from the task after application of the method. In general, we annotate conclusions in backward application directions with "−". However, it might ocassionally make sense to remove the instantiation of a premise from a task when we apply a method in forward direction. Consider for example a situation where we apply a method in forward direction and want to cancel one of the used premises afterwards. This could for instance be the case, when we use a method to simplify a premise $3 + x + 12$ with the help to $x + 15$[10]. In this situation we might want to get rid of the initial premise because we now have obtained an equivalent, but simplified version.

The "∗" label causes a default behavior of the argument. Arguments labeled with "*" must be instantiated in the respective method matching and remain in the task after application of the method.

The next step in the specification of an application direction are the application conditions. Application conditions are arbitrary *LISP*-expressions that must all evaluate to *true* in order for the method to be applicable in that direction. Using application conditions we can encode local constraints as to when to apply the method in the particular direction represented by the application direction.

(application-condition <lisp-expr>)

Eventually, outline computations < comp > can be specified to bind methods variables. Outline computations can be arbitrary LISP expressions that may even return multiple values. These values will be assigned to the method variables in <val-list>.

```
<comp-list> ::= <comp> | ε
                <comp> <comp-list>
<comp>      ::= ( <val-list> <lisp-expr>)
```

6.8.3 Specifying $\subseteq -TRANS$

In this section we step through the specification of our already familiar sample method $\subseteq -TRANS$ to see how the language introduced above is used to specify concrete methods in CORE. We first define the method that describes the inference step $\subseteq -TRANS$ before we turn to the specification of the application directions.

Let us start with the declarations part of the method. Because we want to apply the method to sets of arbitrary type, we declare a type variable `aa` and variables `a`, `b` and `c` for sets of type `(o aa)` which we will use to describe the formula scheme of our method.

[10]In Such methods can be realized by calls to external computer algebra systems

```
(declarations
    (type-variables aa)
    (variables (a (o aa)) (b (o aa)) (c (o aa))))
```

Next, we need to introduce the arguments of the method. Our method has three arguments p_1, p_2 and c and no additional arguments. The formula scheme for the arguments is given in 6.2. Accordingly, we declare the arguments p_1, p_2 and c as follows (we assume that `subset` is already declared in theory *naive-set*).

```
(premises (P1 (subset a b)) (P2 (subset b c)))
(conclusions (C (subset a c)))
```

What is left to specify are the outline mappings of the method. We want to apply the method in three directions. Forward (p_1 and p_2, but not c are instantiated) and in two different backward directions (c and p_2 but not p_1 instantiated, and c and p_1 but not p_2 instantiated). We do this by introducing the application directions `subsettrans-m-f`, `subsettrans-m-b1` and `subsettrans-m-b2` respectively. This yields the outline mapping

```
(outline-mappings
   (((nonexistent existent existent) subsettrans-m-f)
    ((existent existent nonexistent) subsettrans-m-b1)
    ((existent nonexistent existent) subsettrans-m-b2)))
```

Eventually, we specify the CORE tactic that we use to expand the method.

```
(expansion subsettrans-m-exp)
```

In fact, `subsettrans-m-exp` is a CORE tactic that implements the algorithm on page 59. Putting the pieces together yields the following method specification.

```
(defmethod subsettrans
       (in naive-set)
        (declarations
        (type-variables aa)
        (variables (a (o aa)) (b (o aa)) (c (o aa)))
        (parameters )
        (premises (P1 (subset a b)) (P2 (subset b c)))
        (conclusions (C (subset a c)))
        (outline-mappings
            (((nonexistent existent existent) subsettrans-m-f)
             ((existent existent nonexistent) subsettrans-m-b1)
             ((existent nonexistent existent) subsettrans-m-b2)))
        (expansion subsettrans-m-exp)
        (manual "")
        (help "")))
```

We still need to define the application directions for the method. For this particular sample method this is rather easy. We only consider the forward direction. application directions for the other directions can be specified analogously.

Because we do not want to restrict forward application of the method we can leave the application-condition empty (the empty application direction is equivalent to *true*). Similarly, we do not need any additional computations. Hence, we merely need to annotate premises and conclusions of the method with $+$, $-$ and $*$. The conclusion c has to be added to the support windows of the task and is labeled accordingly with $+$. Because we want instantiations of the premises to remain in the task after application of the method, we annotate p_1 and p_2 with $*$. At the end, we obtain the application direction

```
(defdirection subsettrans-m-f
              subsettrans-m
              (premises (* P1) (* P2))
              (conclusions (+ C))
              (application-conditions
               )
              (outline-computations
               )
              (manual "")
              (help ""))
```

6.9 Chapter Summary

In this chapter we started with a comparison of the tactics and methods of the ΩMEGA system. We have seen that they have much in common; i.e. they are both declarative specifications of inference steps. We have therefore argued that these methods and tactics can in principle be subsumed under a uniform datastructure.

We then introduced a notion of a method for the CORE system which can be used for interactive as well as for automated proof planning. When we developed the concept of a method we tried to separate the inference step described by a method from the different application directions of that inference step. This distinction between a method and its application directions is also reflected in the specification language that we introduced to be able to define methods for the use in CORE.

So far we have not been concerned with finding putative instantiations for a methods arguments. In the remainder of this thesis we will adapt ΩMEGAs concurrent suggestion mechanism ΩANTS to CORE which will then be able to identify methods that are applicable in a given proof state and also suggest instantiations of the methods arguments. Before describing how ΩANTS was modified and adapted to CORE we describe the mechansim in some detail.

Chapter 7

The ΩANTS suggestion mechanism

In the previous chapters we have worked on making the interactive use of CORE more user-friendly. We have introduced the concept of tasks to make proper use of COREs strength in applying assertions. Furthermore, we introduced methods which enable us to perform abstract proof steps. However, let us think about how we can further support the interactive use of the system. Three issues make interactive proof construction in CORE still difficult.

The first problem is concerned with the application of replacement rules. Having selected a replacement rule for application, we have to identify the term position inside the active window at which we want to apply the rule. When the active window contains a large formula this can become quite difficult. Furthermore, although the system now heuristically selects replacement rules for application of a particular assertion (cf. Sec. 5.1) this choice might sometimes be incorrect. In this case we have to retreat to checking all admissible replacement rules for applicability. We have argued earlier that this can become quite tedious.

A similar problem holds for methods. Because the number of methods can easily become large, it is difficult to keep track of all methods available in the system, let alone to determine which of the methods are applicable. This points us directly to the third problem. Assume that we have selected a method for application. How do we instantiate the arguments of the method? Remember that in the case of CORE we do not instantiate methods with proof lines that can easily be referenced. Rather, we instantiate arguments with subformulas inside the windows of a given task. Identification of putative instantiations is hence not trivial.

This motivates the need for a suggestion mechanism that automatically checks which methods or replacement rules are applicable in a given proof state and then suggests these rules and methods for application. Ideally, the mechanism is also able to provide suggestions of how to instantiate the arguments of an applicable method. In the ΩMEGA system this job is handled by the multi-agent architecture ΩANTS (see [BS98, BS00] and [BS99b] for an extension of the system). This architecture is based on societies of agents which compute possible argument instantiations for ΩMEGAs inference rules in the time between two user interactions. These argument instantiations are then heuristically ordered and most promising instantiations are dynamically presented to the user as suggestions about how to continue the proof.

The ΩANTS mechanism is in principle calculus independent and it should therefore be no problem to adapt it to the CORE system. In this chapter we will introduce the mechanism in more detail before the next chapter describes how the mechanism can be adapted to support interactive reasoning in CORE.

Before turning to the description of the ΩANTS system it is necessary to characterize the inference rules that can be supported by ΩANTS.

7.1 Supported Inference Rules

ΩANTS computes argument instantiations for inference rules that consist of premises $p_1, \dots, p_m$, conclusions $c_1, \dots, c_n$ and possibly additional parameters $\pi_1, \dots, \pi_k$ that are associated with formula schemes $P_1, \dots, P_m$, and $C_1, \dots, C_n$ which constrain possible instantiations of the arguments. That is, the inference rules that can be supported by ΩANTS are of the following general form

$$\frac{p_1 : P_1 \quad \dots \quad p_m : P_m}{c_1 : C_1 \dots c_n : C_n} \mathcal{R}(\pi_1, \dots, \pi_k)$$

Note that this general form comprises ΩMEGA tactics and calculus rules as well as the methods that we introduced for the CORE system in Chapter 6.

It is typical for this type of inference rules that there are strong *syntactical dependencies* between the arguments of a rule. These dependencies manifest themselfs in two ways: Firstly, actual argument instantiations for some arguments of a rule put syntactic constraints on possible instantiations of other arguments. Secondly, when some arguments are left uninstantiated, these arguments have to be computed from already given instantiations of arguments by a pattern matching process. This is best illustrated by using the $\Rightarrow_E$ rule of the ΩMEGA system as an example. The argument pattern of the $\Rightarrow_E$ rule is the following

$$\frac{p_1 : A \quad p_2 : A \Rightarrow B}{c : B} \Rightarrow_E \tag{7.1}$$

Here one finds the following syntactical constraints: the argument p_1 has to be instantiated with a formula A which is syntactically equal to the antecedent of the implication $A \Rightarrow B$ that serves as an instantiation of $< p_2 >$. A similar dependency holds between the argument instantiation $< c >$ for c and the succedent of p_2. Hence, instantiating some arguments of an inference rule restricts the set of putative instantiations for the remaining arguments.

In ΩANTS we account for this dependencies by specifying agent societies for each rule. Each agent in a society is specialized to search for instantiations of one particular argument of an inference rule by taking into account instantiations of other arguments that are already present. This has the advantage that the order in which instantiations for these arguments are provided does not matter, because it is possible to specify one agent for each combination of existing/nonexisting instantiations.

7.1.1 Partial Argument Instantiation (PAI)

The concept of a *partial argument instantiation* (PAI) is crucial for the understanding of the ΩANTS system. When speaking about partial argument instantiations we always have in mind a mapping from the arguments of an inference rule to actual instantiations of these arguments. In general, arguments need to be only partially instantiated for an inference rule to be applicable. In this sense, PAIs are exactly the method matchings introduced in Chapter 6. In the context of the CORE system we agree to use the terms PAI and method matching synonymously.

The main computational task of ΩANTS is to compute most complete partial argument instantiations for each inference rule in a given proof state. Since only rules for which a non-empty PAI exists are applicable, this already allows us to filter out rules which cannot be applied in a given situation.

Computation of partial argument instantiations is done by societies of agents that concurrently gather information about the current proof state. These agents work in the background of the theorem proving process in the time while the user remains inactive.

The choice for a concurrent computation model is mainly motivated by two considerations: Firstly, computation of some arguments might involve some undecidable problems such as higher-order unification. In a sequential computation this could prevent some computations from being executed. Secondly, some computations might be computationally less expensive than others and can therefore be expected to deliver a result earlier in time. The use of concurrent computations makes it possible to present the result of the agents computations dynamically to the user. Argument instantiations that can be quickly computed will be reported immediately to the user while computationally expensive processes deliver their results with a respective delay. The dynamic presentation of suggestions does not force the user to wait until all computations are finished. Rather he can apply a suggestion at any time, where the quality of the suggestions usually improves the more time is given to the agents. Distribution of computations and the anytime character of the computations are the main strengths of the ΩANTS mechanism.

When we now describe the ΩANTS architecture we do this with respect to the ΩMEGA system to see more clearly where the current implementation has to be changed in order to support proof construction in CORE.

7.2 The Architecture

The ΩANTS mechanism is essentially a three layered blackboard architecture (see Figure F 21, page 87). At the lowest level, societies of agents concurrently compute PAIs for the available inference rules. At the second and third layer these PAIs are then sorted according to certain heuristics in order to present only the most promising suggestions (these are currently the most complete PAIs) to the user.

We now introduce the architecture level by level, starting at the lowest level, at which the core computations take place.

7.2.1 Argument Agents

The core computations of the ΩANTS mechanism are executed in parallel by societies of so called *argument agents* (depicted in Red in Figure F 21). One such agent society is associated with each inference rule that is supported by the suggestion mechanism. Within a society, agents cooperatively try to compute a most complete PAI for the respective inference rule. To be able to do so, every argument agent within a society is specialized for providing instantiations of a particular argument of the inference rule. Computations of actual arguments are in general dependent on information contained in already instantiated arguments. Every argument agent therefore takes a PAI as input and tries to instantiate a further argument of that PAI. In order to characterize the computations performed by the argument agents we distinguish between three kinds of agents.

The first class of argument agents comprises agents that search the support lines of ΩMEGAS proof-data structure for putative instantiations of their associated argument (i.e. a premise). Those agents are called *support agents.*

Agents working for conclusions of an inference rule perform their search amongst the open nodes of the current proof state. Agents of this type are hence called *goal agents.*

The computations of the individual argument agents are a simple search through the open- and support lines of a proof state. For every such line an argument agent decides whether the formula of this line matches the corresponding scheme of the associated argument. If a formula matches the formula scheme and does not conflict with already existing instantiations of other arguments then the corresponding proof line will be suggested as an instantiation for the respective argument. Technically the computations are performed with the help of a *Lisp*-predicate which has to be specified for each argument agent. This predicate will be applied to putative instantiations and returns *true*, if the respective node is an instantiation for the corresponding argument. Hence, goal and support agents are also referred to as *predicate agents.*

A third kind of argument agents does not search for instantiations of premises or conclusion, but simply computes instantiations for the additional arguments of an inference rule (e.g. computation of a most general unifier). These agents will be called *function agents* accordingly and are equipped with a function rather than a predicate.

7.2.2 Specifying Argument Agents

Argument agents are the only part of the architecture that has to be specified by the user. The remaining components are generated automatically by the system. Argument agents are specified in a *Lisp*-like language. At this point, we provide a brief, informal description of the language, which is followed by an example for an agent specification.

The keywords `s-predicate`, `c-predicate` and `function` are used to specify the type of an agent; i.e. `s-predicate` denotes a support agent while `c-predicate` introduces a goal agent. Not surprisingly, function agents are denoted by `function`. The remaining specification of an agent consists of two parts. The first part defines a *dependence, goal* and *exclusion* set which specify which information must be present in a PAI for the agent to be applicable on it. These three sets are the following:

- **Dependence Set:** Arguments in the dependence set of an agent have to be already instantiated in a PAI in order for the agent to be applicable (see above).
- **Exclusion Set:** contains names of all arguments of a command that must not be instantiated for the agent to be applicable on the respective PAI.
- **Goal Set:** Contains all arguments for which the agent tries to compute an instantiation.

Dependence set, exclusion set and goal set have to be mutually disjoint. The union of these sets however has to be the complete argument set of the inference rule. By specifying these sets for each argument agent we can make sure that agents only work on PAIs which contain the information that is relevant for their computations. Goal, dependence and exclusion set are declared with the help of the keywords `goal`,`uses` and `exclude` respectively (cf. Figure F 17).

The second part of an agent specification is a *Lisp*-predicate (for goal and support agents) or function (in the case of function agents) for the agent, which determines the actual computations performed by the agent.

Sample Agent

As an example of an argument agent let us consider one of the agents that search for an instantiation of the premise p_2 of the $\Rightarrow_E$ rule. Looking at the $\Rightarrow_E$ rule (page 69), we can see that any instantiation of the argument p_2 has to be an implication line whose succedent is syntactically equal to the actual conclusion argument. A definition of an argument agent that searches for instantiations of the premise p_2 is depicted in Figure F 17 below.

The depicted agent is applicable on PAIs where the conclusion c (cf. *uses*-slot) but not the premises p_1 and p_2 are already instantiated. If the agent finds

```
(s-predicate (for p2)
             (uses c)
             (exclude p1)
             (definition (impe-p2.pred p2 c)))
```

Figure F 17: Specification of a support agent for the $\Rightarrow_E$ rule

such a PAI $P =< c :< c >, p_1 : \epsilon, p_2 : \epsilon >$ it applies the *impe-p2.pred* predicate to all support lines of $< c >$. The predicate returns *true* for all those support lines that are labeled with an implication whose succedent is equal to the already existing instantiation $< c >$ of the conclusion c. For every such line the agent returns a copy P' of P where p_2 is now instantiated with the respective proof line.

Example: Let us consider a proof state

$$\begin{array}{lll} L_1 & \vdash P(x) \Rightarrow A(x) & (Ax) \\ L_2 & \vdash R \Rightarrow T & (Ax) \\ L_3 & \vdash M(a,b) \Rightarrow A(x) & (Ax) \\ G & \vdash A(x) & (Open) \end{array}$$

and an already existing PAI $< c : A(x) >$ for the $\Rightarrow_E$ rule. Our agent further completes this PAI by suggesting L_1 and L_3 as instantiations for p_2. It will therefore create two new PAIs

$$< c : A(x), p_2 : P(x) \Rightarrow A(x) >, < c : A(x), p_2 : M(a,b) \Rightarrow A(x) >$$

A complete example of how the agents interact in the computation of PAIs is given below. Let us first turn to the next layer of the blackboard architecture.

7.2.3 Command Blackboard

To enable argument agents to cooperatively compute most complete PAIs the argument agents for a command must have a means to communicate, i.e. to exchange partial results (PAIs). This is done via so called *command blackboards*. Agents of an agent society have access to a command blackboard on which they write the results of their computations. Furthermore, each argument agent constantly monitors its command blackboard in order to detect new entries (PAIs) on which it is applicable. Most complete PAIs are therefore computed as follows. Initially, every command blackboard contains only the empty PAI. This triggers an initial agent (with an empty dependence set) which extends the empty suggestion by providing new instantiation for the arguments in its goal set. The resulting PAIs are then written onto the blackboard where they trigger other agents in turn.

7.2.4 Sample Computation for $\Rightarrow_E$ command

To demonstrate how the argument agents cooperatively compute PAIs for their associated command, we consider a sample computation for the $\Rightarrow_E$ command. We restrict the example to the subset of argument agents that compute PAIs for backward application of the $\Rightarrow_E$ rule. To apply the rule backwards we always need an instantiation of the conclusion c. Hence, the initial agent must have c in his goal set, whilst the dependence set and the exclusion set are empty. This agent is denoted as $\mathcal{A}^{\{c\}}_{\{\}\{\}}$. Since this agent works on the initial (empty) suggestion it does not need to check whether possible instantiations for its goal-argument syntactically match already existing instantiations of other arguments. Accordingly, the agent will simply suggest any open node as a possible instantiation for the conclusion of the associated rule.

Apart from this goal agent, further agents are required to search for appropriate instantiations of the arguments p_1 and p_2. This will be done by two agents as follows: One agent $\mathcal{A}^{\{p_2\}}_{\{c\}\{p_1\}}$, tries to further complete PAIs where the conclusion, but not p_1 is instantiated. This agent has to consider all available support nodes and suggest those nodes as instantiations for p_2 that i) are labeled by a formula which is an implication and ii) of which the succedent term is syntactically equal to the instantiation of c.

A third agent $\mathcal{A}^{\{p_1\}}_{\{c,p_2\}\{\}}$ tries to find instantiations for the premise p_1. It takes suggestions (PAIs) with instantiated conclusion and premise p_2 and searches for a support node which is labeled with a formula that matches the already present $< c >$ and $< p_2 >$.

We now study the interplay between this agents at hand of the following proof situation:

L_1	$\vdash P(a)$	(Ax)
L_2	$\vdash P(a) \Rightarrow Q$	(Ax)
L_3	$\vdash A \Rightarrow S$	(Ax)
L_4	$\vdash R(x,y) \Rightarrow Q$	(Ax)
G_1	$\vdash Q$	$(Open)$

When the suggestion mechanism is initialized, all command blackboards, in particular the one for the $\Rightarrow_E$ rule, contain only the empty PAI $<>$. This triggers computation of agents with an empty dependence set. In our example, the only agent with an empty dependence set is $\mathcal{A}^{\{c\}}_{\{\}\{\}}$. It suggests every open goal as an instantiation for the conclusion c. In the above example G_1 is the only open line, so the agent instantiates c with G_1 and writes the PAI $< c : Q >$ onto its command-blackboard.

This in turn will set of agent $\mathcal{A}^{\{p_2\}}_{\{c\}\{p_1\}}$ which searches the support lines for nodes with an implication of which the succedent is equal to the instantiation of c. In our case these are lines L_2 and L_4. Hence, the agent returns the following extensions of the PAI: $< c : Q, p_2 : P(a) \Rightarrow Q >$ and $< c : G_1 : Q, p_2 : R(x,y) \Rightarrow Q >$, which are in turn added to the command blackboard. As a consequence the command blackboard now contains the entries shown in Figure F 18.

$(1) < c : G_1 : Q, p_2 : L_2 : P(a) \Rightarrow Q >$
$(2) < c : G_1 : Q, p_2 : L_2 : R(x,y) \Rightarrow Q >$
$(3) < c : Q >$
$(4) <>$

Figure F 18: Content of the command blackboard for the $\Rightarrow_E$ rule

Eventually, $\mathcal{A}^{\{p_1\}}_{\{c,p_2\}\{\}}$ detects entries (1) and (2) in Figure F 18 on which it is applicable. For the first of the applicable entries (i.e. entry 2 in Figure F 18) it cannot find a support line whose formula equals the antecedent of line L_2. However, the second PAI (entry 1 in Figure F 18) can be extended by line L_1 as instantiation of p_1. Hence, the agent $\mathcal{A}^{\{p_1\}}_{\{c,p_2\}\{\}}$ adds a complete PAI: $< c : G_1 : Q, p_1 : L_1 : P(a), p_2 : L_2 : P(a) \Rightarrow Q >$ to the command blackboard that now contains the entries depicted in Figure F 19

$(0) < c : G_1 : Q, p_1 : L_1 : P(a), p_2 : L_2 : P(a) \Rightarrow Q >$
$(1) < c : G_1 : Q, p_2 : L_2 : P(a) \Rightarrow Q >$
$(2) < c : G_1 : Q, p_2 : L_2 : R(x,y) \Rightarrow Q >$
$(3) < c : Q >$
$(4) <>$

Figure F 19: Command Blackboard for $\Rightarrow_E$ rule after all argument agents have finished their computations.

7.2.5 Command Agents

On top of the layer constituted by the argument agents and command blackboards we find the so called *command agents* (Light Red in Figure F 21). One such command agent is associated with each command blackboard. The job of a command agent is to monitor its command blackboard and to order the incoming entries from most to least promising. The heuristic which is used to sort the entries can be declaratively specified and is therefore easily exchangeable. Currently, command agents prefer more complete PAIs (entries) over entries with more uninstantiated arguments.

Looking at the sample computation above, the command agent for the $\Rightarrow_E$ rule would rate the complete PAI on the blackboard in Figure F 19.

$$< c : G_1 : Q, p_1 : L_1 : P(a), p_2 : L_2 : P(a) \Rightarrow Q >$$

as most promising and therefore propagate it to the suggestion blackboard.

7.2.6 Suggestion blackboard and suggestion agent

The *suggestion agent* together with its suggestion blackboard constitutes the highest level in the three-layered architecture. The purpose of suggestion agent and suggestion blackboard resembles those of the command agents: the suggestion agent monitors the suggestion blackboard where the pre-selected suggestions which are computed at the lower layers of the architecture accumulate. Similarly to the command agents the suggestion agent keeps the entries on the command blackboard heuristically ordered. Again, those PAIs are preferred that are most complete. If two entries are equal to that respect, the entry for the inference rule that is assumed to represent the larger proof step is preferred. To be able to make this judgments the suggestion agent has some notion of the *expansion size* of an inference rule (see [BS98]).

The suggestion blackboard is the only blackboard that the user can access directly. Because this blackboard contains only the "best" suggestions from each command blackboard, he always finds the most promising suggestions on the suggestion blackboard, with the highest ranked suggestion at the top. He can select entries from the suggestion blackboard for application at any time. When an entry is selected the inference rule associated with the PAI is applied with the arguments provided by the PAI.

A change of the proof-state (as is for instance caused by the application of a suggestion) leads to a reset of the whole system. During this reset, all blackboards are erased and initialized with the empty suggestion. The agents of the mechanism are then able to compute suggestions for the new proof state

7.3 Chapter Summary

In this chapter we introduced the three layered blackboard architecture ΩANTS which is used as a suggestion mechanism in ΩMEGA. We have seen how agents at the lowest layer cooperatively compute PAIs for ΩMEGAS inference rules. These suggestions (PAIs) are then heuristically sorted before the most promising suggestions are presented to the user. It was also demonstrated how the argument agents at the lowest layer are specified.

In the remaining chapter we will try to adapt the ΩANTS mechanism to the CORE system. The key problem will be to support the dynamically generated replacement rules for which no agents can be specified in advance.

Chapter 8

ΩANTS and CORE

Throughout the previous chapters we have seen that interactive proof construction in CORE can benefit enormously from a suggestion mechanism that automatically checks for applicable methods and replacement rules and which makes suggestions about which inference step to apply next. We have introduced the ΩANTS mechanism which performs these tasks in the ΩMEGA system. In this chapter we describe how this mechanism was adapted to CORE. To do so we first describe the exact nature of the computations that ΩANTS has to perform in the CORE system and work out how they differ from corresponding computations for the ΩMEGA system. Thereafter we describe the changes of the ΩANTS system that are implied by this analysis.

8.1 Computations

We have already said that we want to support the application of replacement rules and methods in CORE with the help of ΩANTS. However, the computations necessary to check for the applicability of both objects are slightly different. We therefore first analyze the requirements to support replacement rule application with ΩANTS before we turn to method application.

8.1.1 Replacement Rules

Adaptation of ΩANTS to support replacement rule application is more difficult than mere support of method application. The reason for is that the replacement rules that occur during a proof attempt are not known in advance and hence, we cannot specify agents societies for the individual rules. As a first solution of this problem we will devise a generic *rule agent* that checks for the applicability of all admissible replacement rules for a given task in a sequential manner. Of course one could generalize this approach and specify multiple of these rule agents such that each agent works for a different class of rules. However, it does not seem possible to easily avoid sequential processing when dealing with the dynamically generated rules. A possible way out is alluded to in [VBA03] who suggests to employ a generic agent for every context-formula, which computes suggestions as to how to apply the respective formula to the goal.

The question that concerns us is, what are the computations that the rule agent has to perform? Given a replacement rule $\mathcal{R} = i \rightarrow< v_1^{p_1}, \ldots, v_n^{p_n} >$ and a task $T = \Sigma \rhd G$ we are interested in the following information

1. Are there subterms $t_1, \ldots, t_k$ in G on which $\mathcal{R}$ is applicable and if so, what are the t_i ? In fact we are interested in the position π_i of t_i within G such that $t_i = G_{|\pi_i}$ unifies with i under a substitution σ_i. This information is in particular sufficient for deciding whether $\mathcal{R}$ is applicable wrt. T; i.e. $\mathcal{R}$ is applicable if at least one such t_i exists.

2. We know that each $v_i^{p_i}$ of $\mathcal{R}$ will become the goal window of a new task. However, if we can already find an unconditional $v_i^{-p_i}$ in the support windows Σ of T then we can prove the respective $v_i^{p_i}$ immediately by application of the rule $v_i^{p_i} \rightarrow \mathit{true}^+$ which we obtain from $v_i^{-p_i}$. This is analogous to the closure of the premises of a method in Chapter 6.

To illustrate the just said, consider a task

$$Q(a)^-, R(a)^-, R(b)^-, (R(x^\gamma) \Rightarrow P(c))^- \rhd P(c)^+ \wedge Q(x^\gamma)^+$$

This somewhat artificial example is well suited to characterize the computations we are interested in. What information do we want our rule agent to compute for this task? First, we are interested in the replacement rules that are applicable to the goal window $P(c)^+ \wedge Q(x^\gamma)^+$ of the task. Amongst others, this is the rule $\mathcal{R}_0 = P(c)^+ \rightarrow< R(x^\gamma)^+ >$, which can be applied to $P(c)^+$ at position $\pi = [1]$ in the goal window (cf. 1, above) . We are then interested in possible instantiations for the value $R(x^\gamma)^+$ of rule $\mathcal{R}_0$ (cf. 2, above). Candidate instantiations are $R(a)^-$ and $R(b)^-$. Here we have arrived at an important problem. We cannot simply instantiate values v of a replacement rule with any occurrence o that unifies with v. The reason is that we might instantiate variables in other subformulas in a way that prevents us from deriving our initial goal. With respect to this example we can observe, that $R(b)^-$ would be the wrong choice for an instantiation of $R(x^\gamma)^+$, because of the variable x^γ that $R(x^\gamma)^+$ shares with $Q(x^\gamma)^+$. If we were to first apply $\mathcal{R}_0$ to $P(c)^+$ and then $R(b)^+ \rightarrow \mathit{true}^+$ to the newly introduced $R(x^\gamma)^+$, we would transform our task into

$$Q(a)^-, R(a)^-, R(b)^-, (R(x^\gamma) \Rightarrow P(c))^- \rhd \mathit{true}^+ \wedge Q(b)$$

which we cannot close any more. In short, our problem consists of the following: when we close some of the subgoals (values) that are introduced by application of a replacement rule automatically, this corresponds to the execution of further proof steps. Of course, this might involve a choice as to which proof steps to carry out. A wrong choice will require backtracking which we want to do as rarely as possible in interactive proof construction.

The problem is obviously related to the unifiers σ_i for values v_i of a rule $i \rightarrow < v_1, \ldots, v_n >$ and possible instantiations o_i for these values. We therefore introduce a *unifier selector* S. S is a predicate over substitutions σ_i and we use occurrences o_i as instantiations of values v_i of a rule only if S holds for σ_i. By using different unifier selectors S we can implement various heuristics for the selection of value-instantiations. Currently we have chosen S in a way that S holds for a substitution σ only if σ is empty or σ instantiates only variables that occur nowhere else in the FVIFT for the current proof state. With this choice of S we have implemented a *least commitment* strategy and can be certain that we only instantiate variables automatically if this does not affect the construction of proofs for other subgoals.

We now introduce the concept of a *rule matching* in analogy to a method matching to describe the computations that the rule agent has to perform.

Definition 8.1.1 *(Rule Matching) Let $\mathcal{R} =< v_1^{p_1}, \ldots, v_n^{p_n} >$ be an admissible replacement rule for the window G of a task $T = \Sigma \triangleright G$ and S a unifier selector. If R is the smallest subtree of the current* FVIFT *that contains all windows in T then the rule matching for $\mathcal{R}$ with respect to T is a partial mapping*

$$match_{\mathcal{R}} : \{i, v_1, \ldots, v_n\} \rightarrow OCC_T$$

where OCC_T is the set of all occurrences inside the windows in $\Sigma \cup \{G\}$ and the following conditions hold for $match_{\mathcal{R}}$.

1. *If $match_{\mathcal{R}}(i^p) = o^q$ then $p = 0$ or $p = q$ and o^q lies inside G.*
2. *If $match_{\mathcal{R}}(v_i^{p_i}) = o^q$ then*
   (a) *if $p_i \neq 0$ then $p_i \neq q$.*
   (b) *If $p_i = 0$ then $q \neq \tilde{p}$ where $\tilde{p}$ is the polarity of $match_{\mathcal{R}}(i^p)$.*
   (c) *o^q is unconditional in R and lies in one of the support windows Σ of T.*
3. *If $\sigma_k = unify(match_{\mathcal{R}}(v_k), v_k)$ for $k = 1, \ldots, n$ and $\sigma = unify(match_{\mathcal{R}}(i), i)$ then S holds for σ and each of the σ_i, $i = 1, \ldots, n$.*

Similarly to method matchings we refer to an o with $match_{\mathcal{R}}(v_i) = o$ as an instantiation of v_i and write $< v_i >= o$. As we did with method matchings we often represent a rule matching $match_{\mathcal{R}}$ of a rule $\mathcal{R}$ as $< i :< i >, v_1 :< v_1 >, \ldots, < v_n >>_{\mathcal{R}}$.

Point 2) of Definition 8.1.1 needs some explanation. We have defined a rule matching in a way that the instantiations for the values v_i of a rule $\mathcal{R}$ can be used to immediately prove subgoals that are introduced through application of $\mathcal{R}$. If $\mathcal{R}$ is a rewriting replacement rule then the polarity of i and v_1 is undefined (i.e. 0). However, the subgoal that corresponds to the value v_1 will inherit the polarity of the occurrence to which $\mathcal{R}$ is applied; i.e. $match_{\mathcal{R}}(i)$. Therefore, instantiations for v_i must be of opposite polarity as $match_{\mathcal{R}}(i)$ (cf. point 2) in Definition 8.1.1).

The rule agent that we will add to the ΩANTS architecture computes *all* rule matchings for the rules that are admissible for the goal window of the active task. Because only rules $\mathcal{R}$ for which a non-empty rule matching exists are applicable wrt. the active task the rule agent already filters out non-applicable rules.

8.1.2 Methods

The computations necessary to support the application of methods are much easier described than those for replacement rules. We are simply interested in the computation of method matchings for all methods wrt. to the active task. Method matchings therefore play a similar role as the PAIs in the ΩMEGA system. Of course, methods for which no method matching exists are not applicable. Because ΩANTS was designed to compute method matchings (PAIs) we will find that we can easily adapt ΩANTS argument agents to compute method matchings for the CORE system (cf. Sec. 8.2.1).

8.2 Making ΩANTS safe for CORE

Based on the above analysis we are now ready to describe the adaptation of the ΩANTS mechanism to CORE. To be able to deal with the dynamically generated replacement rules for which we want to compute rule matchings we extend ΩANTS by the rule agent that we already alluded to. This rule agent is basically a generic argument agent which computes rule matchings for all replacement rules that are admissible in a given proof state. Along with the rule agent we add a *rule blackboard* and a *rule command agent* to the architecture (cf. Figure F 21). Rule blackboard and rule command agent correspond to command blackboards and command agents respectively and will serve a similar purpose.

Method matchings will be computed with the help of the already existing architecture. Because of the close correspondence between method matchings and PAIs as well as between methods in CORE and tactics in ΩMEGA this part of the architecture needs only little change in order to function in the CORE system.

We now turn to a description of the components of the extended architecture. We begin with those components that are already present in ΩANTS and which are now used to compute method matchings.

8.2.1 Argument Agents

The adaptation of the argument agents to search for method matchings is rather easy. When used in conjunction with ΩMEGA they compute most complete PAIs for the ΩMEGA tactics by searching for argument instantiations amongst ΩMEGAS the proof lines. Now we have to make sure that the search for argument instantiations is performed amongst the occurrences inside the windows of the active task. To make this task accessible to the agents we introduce a new global

```
(s-predicate (for p1)
             (exclude p2)
             (uses c)
             (definition (tac=subset-trans.p1 (:param c) (:param p1))))
```

Figure F 20: Specification of a support agent that searches for possible instantiations of the argument p_1 for the $\subseteq -TRANS$ method. The agent is applicable on method matchings where the conclusion c but not the premise p_2 are already instantiated. `:param` is a keyword that specifies that only the label of the putative instantiation of p_1.

variable `active*task` which always points to the task on the agenda which the user is currently working for. Because we are still essentially searching for formulas that fit into a certain formula scheme we can leave the specification language for argument agents unchanged. Merely the keywords that specify the type of an agent obtain a different meaning.

`s-predicate:` Support agents try to instantiate arguments in the premise of a method. Accordingly they now search amongst the negative occurrences inside the support windows Σ of the active task.

`c-predicate:` Goal agents (or conclusion agents) search for instantiations of a conclusion of a method. This search is now performed amongst the positive occurrences inside the goal window of the active task.

`function:` Function agents compute arbitrary objects as before.

Looking at the specification of an agent that searches for one of the premises of the $\subseteq$-TRANS method in CORE, we see that argument agents can indeed be specified as before. In Figure F 20 the predicate `tac=subset-trans.p1` is the predicate that is applied to all occurrences in the support windows of the active task and which returns *true* for all occurrences that are labeled with a formula which matches the formula scheme $A(\subseteq B)$ for the parameter p_1.

Argument agents compute method matchings in the same cooperative way as they did for PAIs in the ΩMEGA system. This cooperation still depends on the command blackboard that is associated with each agent society.

8.2.2 Command Agents & Command Blackboards

Command blackboards are still monitored by command agents which function exactly as before. As was the case in ΩMEGA the command agents keep the entries on their associated command blackboard in a certain order and propagate best suggestions to the suggestion blackboard. The heuristic that is employed to rank the entries (i.e. method matchings) on the command blackboard prefers more complete method matchings over method matchings with fewer instantiated arguments.

Having seen that the adaptation of ΩANTS for the computation of method matchings is quite straightforward we now concentrate on describing the extension of the architecture which enables the mechanism to support application of replacement rules.

8.3 Supporting Replacement Rule application

We have already argued that because the replacement rules which occur during a proof are not known at the outset of a proof, we cannot specify agents for these rules in advance. The solution we have already sketched earlier on in this chapter is to extend the architecture by a rule-agent that computes rule matchings in a sequential manner. Of course this is only a sub-optimal solution because it breaks radically with the philosophy of ΩANTS to distribute computations over concurrent processes. The most serious drawback of a serialization of computations is that if we want to compute rule matchings for replacement rules $\mathcal{R}_1, \ldots, \mathcal{R}_n$ and get stuck at rule $\mathcal{R}_i$ because we encounter an undecidable problem (e.g. a higher-order unification) or a program error then the system might not recover and we cannot process rules $\mathcal{R}_{i+1}, \ldots, \mathcal{R}_n$ any more. To slightly ease this drawback the rule agent excludes all rules from consideration that contain a higher-order variable (i.e. a variable that ranges over a function or a statement of type o). At the moment we can safely do so because CORE does not yet provide a higher-order logic input language and we are hence restricted to work with first-order problems. The only higher-order formulas that might occur during a proof are some axioms that are automatically loaded (e.g. the induction axiom and the transitivity of $=$) which are only needed in particular domains anyway. Once the system is extended we can introduce a second agent or even several additional agents that exclusively deal with higher-order rules. The benefit of this decision is that we ensure that at least all first-order rules are considered by the rule agent without encountering an undecidable subproblem.

8.3.1 The Rule Agent

Like the argument agents, the rule agent is located at the lowest layer of the architecture. It has access to the windows of the active task where it searches for most complete rule matchings for all admissible replacement rules that can be generated from the support windows $\{w_1, \ldots, w_n\} = \Sigma$ of the actual task. However, recall that we can chose one of the windows w_i to apply the formula contained in this window to the goal window G of the active task (cf. Chapter 4). In this case the rule agent should restrict its search to rules that can be obtained from this particular window. We therefore introduce a second global variable `active*windows` which always contains those windows from which we want to generate and apply rules (that is, it contains either Σ or exactly one $w_i \in \Sigma$; i.e `active*windows` always contains a subset of the supports of the active task).

On a reset, the rule agent computes rule matchings as follows. First, it generates all replacement rules from the $\{w_1, \ldots, w_k\}$ that are denoted by the `active*windows`. For each such rule $\mathcal{R} = i^p \rightarrow < v_1^{p_1}, \ldots, v_n^{p_n} >$ and a task

$T = \Sigma \rhd G$ the agent then generates the sets

$$\begin{aligned} I_{\mathcal{R}} &= \{(o^q,\sigma) | o^q \text{ inside } G, p = q \text{ and } i\sigma = o\} \\ V_{\mathcal{R}}^{v_l^{p_l}} &= \left\{(o^q,\sigma) \,\middle|\, \begin{array}{l} o^q \text{ inside any of the } \Sigma,\ p_l \neq q,\ v_l\sigma = o \text{ and } o \\ \text{is unconditional} \end{array}\right\} \end{aligned}$$

for a resolution replacement rule $\mathcal{R}$ and

$$\begin{aligned} I_{\mathcal{R}} &= \{(o^q,\sigma) | o^q \text{ inside } G \text{ and } i\sigma = o\} \\ V_{\mathcal{R}}^{v_l^{p_l}} &= \begin{cases} \left\{(o^q,\sigma) \,\middle|\, \begin{array}{l} o^q \text{ inside any of the } \Sigma \text{ and } v_l\sigma = o \text{ and } o^q \text{ is} \\ \text{unconditional.} \end{array}\right\} & \text{if } l = 1 \\ \left\{(o^q,\sigma) \,\middle|\, \begin{array}{l} 1.\ o^q \text{ inside any of the} \\ \quad \Sigma \\ 2.\ v_l\sigma = o, p_l \neq q, \text{ and} \\ \quad o \text{ is unconditional} \end{array}\right\} & \text{otherwise} \end{cases} \end{aligned}$$

of instantiations $I_{\mathcal{R}}$ for the input of $\mathcal{R}$ and for the values v_l of $\mathcal{R}$ (i.e. $V_{\mathcal{R}}^{v_l^{p_l}}$). For each replacement rule with $I_{\mathcal{R}} \neq \emptyset$ (i.e. all applicable rules) the rule agent then generates all possible rule matchings from these sets. To see how let

$$\tilde{V_{\mathcal{R}}}^l = \begin{cases} \{(\epsilon, \{\})\} & \text{if } V_{\mathcal{R}}^l = \emptyset \\ V_{\mathcal{R}}^l & \text{else} \end{cases}$$

where ϵ represents the empty instantiation; then each

$$m = ((o^p,\sigma),(o_1^{p_1},\sigma_1),\ldots,(o_k^{p_k},\sigma_k)) \in \tilde{I_{\mathcal{R}}} \times \tilde{V_{\mathcal{R}}}^1 \times \ldots \times \tilde{V_{\mathcal{R}}}^n$$

gives rise to a rule mapping if S holds for σ and each of the σ_i and $\mathcal{R}$ is a resolution replacement rule. For rewriting replacement rules we have to make the additional requirement that $p \neq p_1$ because v_1 inherits the polarity p from the occurrence to which it is applied. The rule matching for $\mathcal{R}$ that corresponds to m is $< i : o, v_1 : o_1, \ldots, v_n : o_n >_{\mathcal{R}}$. Note that as a byproduct of the computation of the $I_{\mathcal{R}}$ we obtain the application positions π_m for each rule matching.

For each of the computed rule matchings $match_{\mathcal{R}}$, the rule agent writes an entry $e = (\pi, match_{\mathcal{R}})$ on the rule command blackboard. This means that each rule matching is augmented with information about where to apply the rule in the goal window of the active task. The entries are then further processed by the rule command agent.

8.3.2 The Rule Command Agent

The rule command agent plays a similar role as the command agents. It monitors the rule command blackboard and sorts the entries on this blackboard heuristically. The best suggestions are then passed on to the suggestion blackboard. The reason why we need an extra agent for the rule blackboard is that the heuristic used by the rule command agent is slightly different to the heuristics used by the command agents; in fact, the heuristics can be declaratively stated and are therefore easily exchanged. By using different types of command-agents we make it possible to use distinct sorting criteria for rule- and method-matchings.

Currently the rule command agent sorts entries $e = (\pi, match_{\mathcal{R}})$ according to the number of values in $\mathcal{R}$ that are not instantiated by $match_{\mathcal{R}}$. Entries with fewer non-instantiated values are preferred over entries with for a matching with a higher number of non-instantiated values. More precisely, the rule command agents orders the entries according to the ordering $\prec_R$, where

$$e_1 \prec_R e_2 \text{ iff } |\{v | match_{\mathcal{R}_1}(v) = \epsilon\}| < |\{v | match_{\mathcal{R}_2}(v) = \epsilon\}|$$

for entries $e_i = (\pi_1, match_{\mathcal{R}_i})$. Entries that are minimal with respect to this ordering are then propagated to the suggestion blackboard.

The motivation for this heuristic is that we propagate those rule matchings to the suggestion blackboard that will introduce the least number of subgoals (recall that each non-instantiated value v in a rule-matching leads to the introduction of a new task for v).

8.3.3 Suggestion blackboard and Suggestion Agent

The suggestion blackboard is the place where the entries from the command agents and the rule command agent accumulate. The suggestion blackboard therefore always contains suggestions as to which method or replacement rules can be applied. Incoming entries are permanently ordered by the suggestion agent such that the most promising entries are always on top of the blackboard. The user can choose an entry for application at any time. When he selects an entry for application it is given to an *application handler* which distinguishes between entries for rule and method matchings.

Entries $e = (match_{\mathcal{M}})$ for method matchings are simply executed as described in Chapter 6 by using $match_{\mathcal{M}}$ as the method matching for $\mathcal{M}$.

When the user chooses an entry $e = (\pi, match_{\mathcal{R}})$ with a rule matching, the application handler applies rule $\mathcal{R}$ at position π to the goal window G of the active task. This leads to a replacement of $G_{|\pi}$ by a subtree

$$\beta(v_{i_1}^{p_{i_1}}, \dots, v_{i_k}^{p_{i_k}}, v_{i_{k+1}}^{p_{i_{k+1}}}, \dots, v_{i_n}^{p_{i_n}})$$

where we already ordered the v_i according to whether they are instantiated in $match_{\mathcal{R}}$ or not. If $v_{i_1}^{p_{i_1}}, \dots, v_{i_k}^{p_{i_k}}$ are the instantiated values of $\mathcal{R}$ in $match_{\mathcal{R}}$ (i.e. $match_{\mathcal{R}}(v_{i_l}^{p_{i_l}}) \neq \epsilon$, for $l = 1, \dots, k$) then we can use the instantiations $< v_{i_l}^{p_{i_l}} >= v_{i_l}^{-p_{i_l}}$ to generate the replacement rule $v_{i_l}^{p_{i_l}} \rightarrow \diamond$ (for $\diamond \in \{true^+, false^-\}$) and apply it to $v_{i_l}^{p_{i_l}}$. We then obtain the simplified subtree

$$\beta(v_{i_{k+1}}^{p_{i_{k+1}}}, \dots, v_{i_n}^{p_{i_n}})$$

Next, a window is opened on each of the remaining $v_{i_l}^{p_{i_l}}$ which become the goal windows of new tasks. Note that we could also create only one new task for the entire formula $\beta(v_{i_{k+1}}^{p_{i_{k+1}}}, \dots, v_{i_n}^{p_{i_n}})$ and leave the decomposition to the user.

8.4 Sample Computation

In this section we go through a sample computation to consider in detail the computations performed by the rule-agent. The example is constructed to show how valid rule matchings are computed and how application of such a rule matching shortcuts multiple basic reasoning steps. We consider a task

$$T = \Sigma, (A(y^\gamma) \wedge B(x^\gamma) \Rightarrow C)^-, A(y^\gamma)^-, B(a)^- \rhd C^+$$

The support windows of T give rise to a number of replacement rules that are admissible for the goal window of the task. Amongst others these are

$$\begin{array}{l} \mathcal{R}_1 = C^+ \rightarrow < A(y^\gamma), B(x^\gamma)^+ > \\ \mathcal{R}_2 = A(y^\gamma)^+ \rightarrow true^+ \\ \mathcal{R}_3 = B(a)^+ \rightarrow true^+ \\ \ldots \end{array}$$

These rules are computed by the rule agent when it encounters the empty suggestion on the rule command blackboard. For each of these rules the rule agent then computes the sets $I_{\mathcal{R}_i}$ of possible application occurrences inside the goal window of T. For our example these sets are $I_{\mathcal{R}_1} = \{(C^+, \{\})\}$ and $I_{\mathcal{R}_i} = \emptyset$ for $i = 2, 3$. This means that only rule $\mathcal{R}_1$ of the above rules is applicable. In a next step the sets $V_{\mathcal{R}_1}^{A(x^\gamma)} = \{(A(y^\gamma)^-, \{\})\}$ and $V_{\mathcal{R}_1}^{B(x^\gamma)} = \{(B(a)^-, \{a/x^\gamma\})\}$ are computed by the rule agent. Note that $(B(a)^-.\{a/x^\gamma)$ is in $V_{\mathcal{R}_1}^{B(x^\gamma)}$ if we assume that the variable x^γ occurs nowhere else in the FVIFT. In this case $S(\{a/y^\gamma\})$ holds because of the way we have chosen S. From the sets $I_{\mathcal{R}_1}, V_{\mathcal{R}_1}^{A(y^\gamma)}$ and $V_{\mathcal{R}_1}^{B(x^\gamma)}$ we obtain the single rule matching

$$match_{\mathcal{R}_1} =< C^+ : C^+, A(y^\gamma)^+ : A(y^\gamma)^-, B(x^\gamma)^+ : B(a)^- >_{\mathcal{R}_1}$$

This rule matching gives rise to an entry $e = ([], match_{\mathcal{R}_1})$ on the rule command blackboard which will then be propagated to the suggestion blackboard. If the user selects entry e for application the suggestion agent passes e to the application handler which applies the rule $\mathcal{R}_1$ to T which leads to the intermediate task

$$\Sigma, (A(y^\gamma) \wedge B(x^\gamma) \Rightarrow C^-, A(y^\gamma)^-, B(a)^- \rhd \beta(A(y^\gamma)^+, B(x^\gamma)^+)^+$$

The handler then immediately generates the rules $A(y^\gamma)^+ \rightarrow true^+$ and $B(a)^+ \rightarrow true^+$ and applies them to the newly introduced occurrences inside the goal window. The task obtained after application of entry e and a simplification of the goal window then is

$$\Sigma, (A(y^\gamma) \wedge B(x^\gamma) \Rightarrow C^-, A(y^\gamma)^-, B(a)^- \rhd true^+$$

8.5 Integration of External Reasoners

In the previous chapters we have identified the need to integrate external reasoning systems, such as automated first-order theorem provers into CORE that try to close inactive tasks automatically. This can now be realized by combining the features provided by the task structure with the ΩANTS mechanism.

To be able to make use of automated provers X such as OTTER or SPASS we assume that we have a method ATP_X for each of these provers (cf. Section 6.6) that can be used to justify a goal by a call to such an ATP in the following way. Argument agents for a method ATP_X instantiate the conclusion of the method with the goal formula of a task. Premises of the method are instantiated with the corresponding support formulas. The agents then invoke the ATP X to see whether the automated reasoner can derive the conclusion from the premises. If this is the case then the corresponding method matching is passed onto the suggestion blackboard.

We want to apply methods ATP_X to inactive tasks, but only if the system load is not too high. This might for instance be the case when most agents have finished their computations, but the user has not yet selected a suggestion.

The idea is now to index every task on the agenda with a natural number i and to augment all blackboard entries with an index for a task. This index will tell the argument agents in which task to search for possible completions of the respective entry (i.e. method- or rule matching). If, in addition, we give the command agents for the methods ATP_X access to the agenda, we can realize the intended application of external reasoners to inactive tasks as follows.

On a reset of the suggestion mechanism, all command blackboards are initialized with the empty suggestion for the active task T_a (i.e. $(a, <>)$). This will set off the agents to search for suggestions as to how to apply their corresponding method to the active task. All command agents can monitor their associated command agents and recognize when the argument agents have finished their computations. If a command agents for a method ATP_X becomes aware that its argument agents have finished their computations for task T_a it adds the empty entry $(b, <>)$ for the next task T_b on the agenda to the blackboard. This will then cause the argument agent for ATP_X to check whether the theorem prover it works for can close task T_b automatically.

An extension of ΩANTS in this way would have the effect that if there is a sufficiently long delay between two user interactions then the agents try to close inactive tasks in the background of the reasoning process. Successful suggestions as to close an inactive task automatically will eventually pop up at the suggestion blackboard along with the suggestions for the current task.

8.6 Chapter Summary

In this chapter we have described how the suggestion mechanism ΩANTS was adapted to the CORE-system. We have seen that this was straightforward with respect to methods. To be able to support the application of replacement rules we had to extend the architecture by a rule-agent.

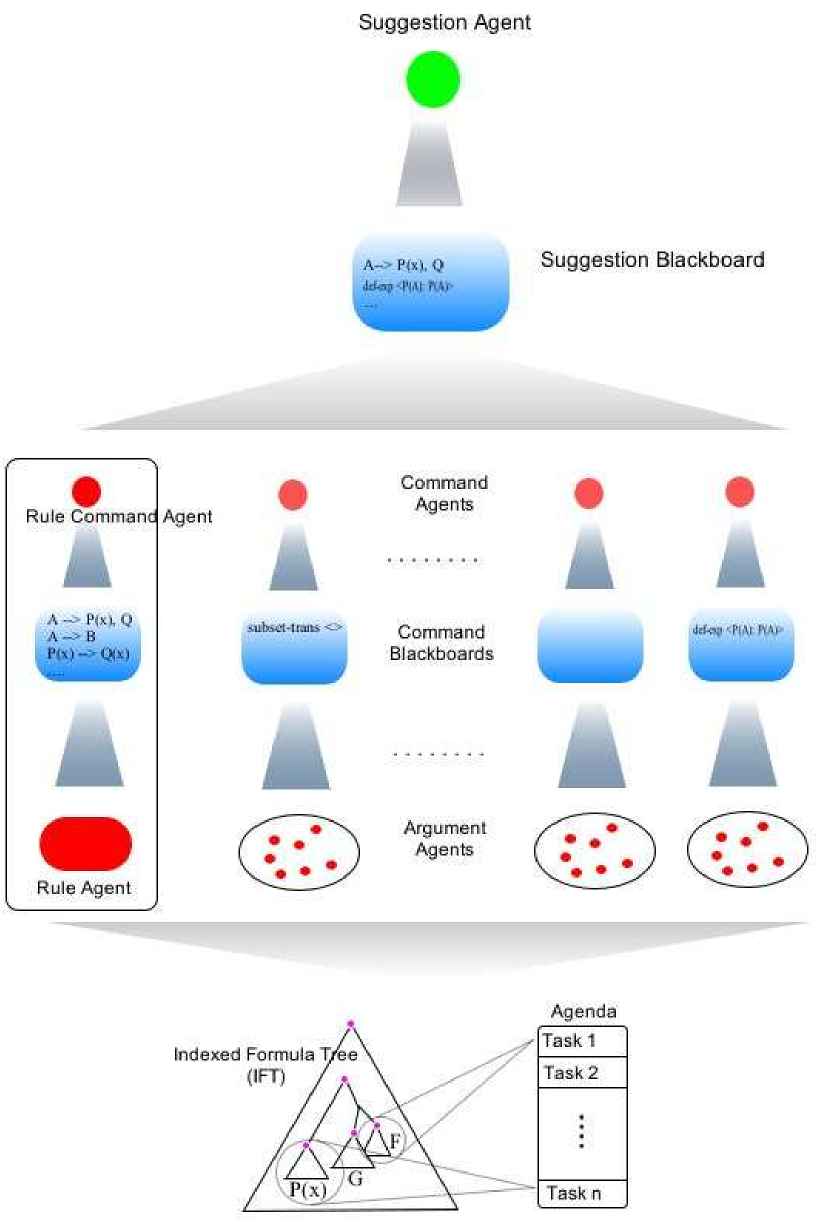


Figure F 21: The extended ΩANTS architecture. Rule Agent, Rule Command Blackboard and Rule Command Agent constitute the extension of the architecture and are found at the left-hand side of the graphic.

Chapter 9

Conclusion & Further Work

In this thesis we were concerned with an extension of the CORE theorem-proving environment by additional functionality to facilitate interactive proof development. To achieve this goal we developed a task datastructure on top of COREs window-inference technique. We further mapped the notion of a proof-method to the CORE system. These methods operate on the task structure and make it possible to perform abstract proof steps in CORE.

Furthermore, we adapted ΩMEGAs suggestion mechanism ΩANTS to CORE where it now supports interactive theorem proving by identifying applicable replacement rules and methods for a given proof state and makes suggestions about which of these inferences to make next.

When we devised the methods for CORE we took into account the experience made with the interactive theorem proving environment ΩMEGA, where distinct datastructures are used to apply abstract inference steps in interactive proof construction and automated proof-planning. We were able to merge the tactics and methods of the ΩMEGA system into a single concept of a method that can now be used in interactive proof development as well as in proof-planning.

The contribution of this thesis therefore lies in the development of a common interface for interactive theorem proving and automated proof-planning which is constituted by the tasks and methods that we have introduced.

Together with the adaptation of the ΩANTS suggestion mechanism we were thus able to provide evidence for two hypothesis. Firstly, it is claimed that CORE provides a well suited basis for human-oriented theorem proving. In Chapter 5 we saw that this is indeed the case if we properly exploit the potential provided by CORE. The implementation of tasks and methods has shown that by using CORE we are able to hide many logical details (such as the quantifier elimination) from the user, while we are at the same time able to use established concepts like methods in the theorem proving process.

Secondly, the successful adaptation of ΩANTS to CORE shows that the agent-based suggestion mechanism ΩANTS is indeed calculus independent as claimed in [BS99b] and can support proof development in various interactive theorem proving environments.

However, although we provided an interface on which interactive theorem proving and automated proof-planning can be based, there is still a way to go in order to make CORE a full fledged interactive theorem proving environment. We now list the main issues that still have to be addressed.

Integration of external reasoning systems Task structure, methods and the ΩANTS suggestion mechanism already provide the basic functionality to integrate external reasoning systems into CORE (see Section 6.6 and Section 8.5). In principle it is now straightforward to invoke such external systems in CORE. A method that uses an automated reasoning system has to translate a subproblem (e.g. a task) into the input language of an appropriate ATP. It should be almost trivial to develop a translation module that, given a problem in CORE representation (i.e. a set of signed formulas), generates a problem uniform description which can be used as input to the mathematical software bus MATHWEB which provides access to a variety of special purpose reasoning systems. A problem that is more serious concerns the expansion of the results returned by the external reasoners (e.g. a proof). Application of inferences that are justified by the call to an external system have to be expanded by translating the result returned by these systems into CORE proof fragments. This requires the development or adaptation of entire translation modules, similar to the TRAMP module [Mei00] which is used in ΩMEGA. However, note that in absence of such a module we can nonetheless make use of external reasoning systems if we trust their results and ignore the problem of expanding their contributions into a CORE proof fragment.

Shared variables We have seen throughout the thesis that variables which are shared between different subformulas might sometimes cause problems, for instance when we want to apply external reasoners to a task that contains variables used in other tasks. This problem requires thorough investigation. A first step into the direction of a solution would be to see how techniques used in other systems that deal with this problem can be used in CORE. For instance it might be worth to pursue a similar approach as in the ΩMEGA system and use a constraint store along with a constraint solver to treat the problem of variable instantiation as a set of constraints.

Automated proof-planning We have completely ignored the problem of automated proof-planning in this thesis. However, the task structure together with the methods that we have introduced should serve as a basis for the development of an automated proof-planner for CORE.

User Interface We have seen that although tasks allow us to present a subproblem in an accessible way to the user it is important that only the information necessary to continue the proof is shown to the user. In fact, the required information might be dependent on various variables such as the proof-strategy preferred by the user (e.g. direct vs. indirect proof), etc.

This issue can be generalized to the broader question of how we can design an interface that takes advantage of COREs novel way to represent a proof-state and to organize the reasoning process. Ideally, the development of such an interface should go together with empirical studies that try to establish which information is considered as relevant by the user (or different types of users) in a given proof situation.

Completeness Of course, when we make use of the task structure introduced in Chapter 4, it is important to establish the completeness of the task manipulation rules. However, a formal treatment of this issue would have been clearly out of scope for this thesis as this would have for instance required a formal definition of the semantics of signed formulas.

Moreover, although we were only concerned with COREs first-order functionality it is essentially a higher-order framework. Once it is possible to load higher-order problems into CORE, any completeness proof has to cover the higher-order case as well. This might not even be possible without new techniques to show completeness of higher-order reasoning frameworks as were only recently proposed in [BBK03].

Term Indexing Currently, the concepts developed in this thesis have only been tested on rather simple theorems from the domain of naive set theory. When the system is to be used for real-world problems we might be able to make the computation processes more efficient by the employment of real term-indexing techniques as we already pointed out in Chapter 5. With term-indexing techniques properly implemented it might for instance be possible to identify applicable replacement rules together with the application position in a more efficient way. However, we postpone work in this direction until we can load higher-order problems in CORE. The reason is that when we use higher-order terms we will encounter new problems, such as how to deal with variable head-symbols of terms. The question of how to employ term-indexing strategies in a higher-order framework is thus a topic on its own.

Appendix A

BNF-Grammar

```
<inference> ::= (defmethod <name>
                   (in <theory>)
                   (declarations
                       (type-variables (<typevar-list>))
                       (variables      (<var-list>))
                       (methodvars     (<mvar-list>)))
                   (parameters (<name-list>))
                   (premises    (<arg-list>))
                   (conclusions (<arg-list>))
                   (outline-mappings (<map-list>))
                   (expansion <name>))

<appl-dir> ::= (defdirection <name> <name>
                  (declarations
                      (type-variables <name-list>)
                      (variables      <var-list>)
                      (methodvars     <mvar-list>))
                  (hypothesis ( <hyp-list> ))
                  (premises ( <arg2-list> ))
                  (conclusions (<arg2-list> ))
                  (application-conditions <list-of-expr> )
                  (outline-computations (<comp-list>))

<arg-list> ::= <arg>
             | <arg> <arg-list>
<arg>      ::= ( <name> <decl_content> ) | ε

<map-list> ::= <mapping> | ε
             | <mapping> <map-list>
<mapping>::= ({existing|nonexisting}* <name> )

<mvar-list> ::= <mvar> | ε
              | <mvar> <mvar-list>
<mvar> ::= ( <name> <mvar-type> )
<mvar-type> ::= <basetype>
               | ( <constructor> <mvar-type> )

<type> ::= <basetype>
         |<type-var>
         |(<type-list>)

<type-list>::= <type> | ε
             |<type> <type-list>
```

```
<typevar-list>::= <type-var> | ϵ
                 |<type-var> <typevar-list>
<type-var> ::=    <name>

<constructor> ::= listof | pairof | ...

<var-list> ::= <var> | ϵ
             | <var> <var-list>
<var>      ::= ( <name> <type> )

<comp-list> ::= <comp> | ϵ
                <comp> <comp-list>
<comp>      ::= ( <val-list> <lisp-expr>)

<val-list> ::= <name> | ϵ
               <name> <val-list>

<hyp-list> ::= <hyp> | ϵ
             | <hyp> <hyp-list>
<hyp>      ::= ( <name> <decl-content> :for <name>)


<arg-list2> ::= <arg2> | ϵ
              | <arg2> <arg-list2>
<arg2>      ::= (+ <name>)
              | (− <name>)
              | (∗ <name>)


<name> = STRING
<theory> ::= <name>


<name-list> ::= <name> | ϵ
               |<name> <name-list>
```

< decl_content > = All terms over all types, variables and function symbols that are visible at this point. These are all objects that have been declared in either the theory, the method or the particular application direction.

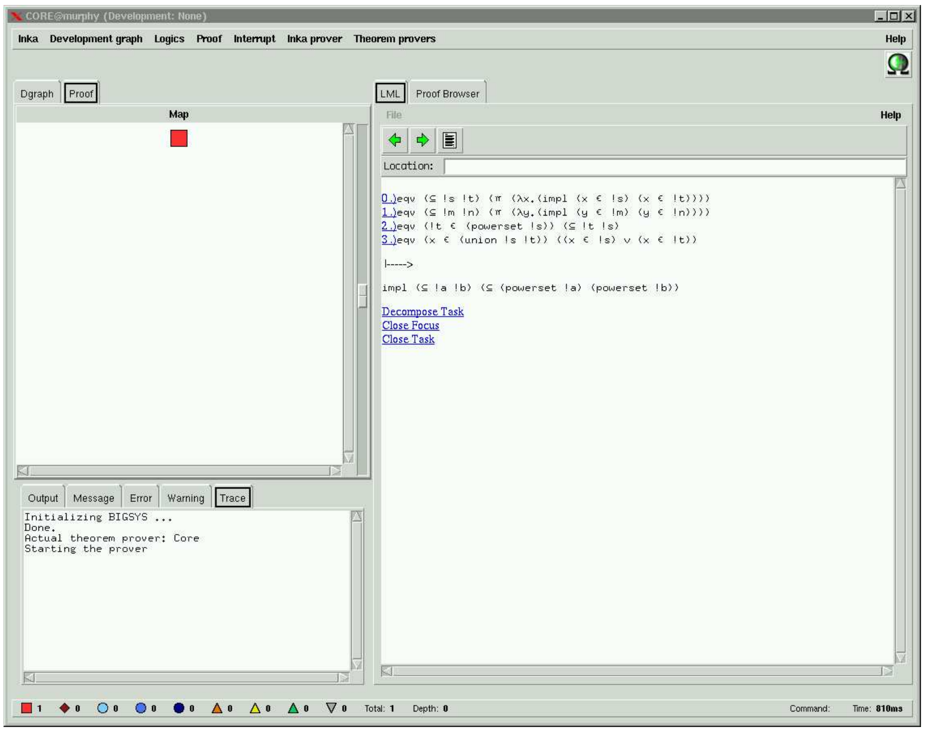
```
<lisp-expr> ::= an arbritrary Lisp-expression

<list-of-expr> ::=<lisp=expr> | ϵ
                | <lisp-expr> <list-of-expr>
```

Figure F 22: BNF grammar for the method specification language introduced in chapter 6

Appendix B

Screenshots



```
0.)eqv (⊆ !s !t) (π (λx.(impl (x ∈ !s) (x ∈ !t))))
1.)eqv (⊆ !m !n) (π (λy.(impl (y ∈ !m) (y ∈ !n))))
2.)eqv (!t ∈ (powerset !s)) (⊆ !t !s)
3.)eqv (x ∈ (union !s !t)) ((x ∈ !s) v (x ∈ !t))

|---->

impl (⊆ !a !b) (⊆ (powerset !a) (powerset !b))

Decompose Task
Close Focus
Close Task
```

```
Initializing BIGSYS ...
Done.
Actual theorem prover: Core
Starting the prover
```

Figure F 23: GUI of the CORE system. The initial task for the example from Chapter 5 is shown in the window on the right-hand side.

Bibliography

[And80] Peter B. Andrews. Transforming matings into natural deduction proofs. In *Proceedingsof the 5th International Conference on Automated Deduction*, pages 281–292. Springer, 1980.

[Aut01] Serge Autexier. A proof-planning framework with explicit abstractions based on indexed formulas. In Maria Paola Bonacina and Bernhard Gramlich, editors, *Electronic Notes in Theoretical Computer Science*, volume 58. Elsevier Science Publishers, 2001.

[Aut03] Serge Autexier. *Hierarchical Contextual Reasoning (Working title)*. PhD thesis, University of the Saarland, 2003. to appear.

[BBC$^+$97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual : Version 6.1. Technical Report RT-0203, 1997.

[BBK03] Christoph Benzmüller, Chad Brown, and Michael Kohlhase. Higher order semantics and extensionality. *In final revision of Journal of Symbolic Logic*, 2003.

[BBS99] Christoph Benzmüller, Matt Bishop, and Volker Sorge. Integrating tps and omega. *Journal of Universal Computer Science*, 5:188–207, 1999.

[BCF$^+$97] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMega: Towards a Mathematical Assistant. In W. McCune, editor, *Proceedings of the 14th Conference on Automated Deduction (CADE–14)*, LNAI, Townsville, Australia, 1997. Springer Verlag, Berlin, Germany.

[Bib82] Wolfgang Bibel. *Automated Theorem Proving*. Vieweg Verlag, Wiesbaden, Germany, 1982.

[BMM$^+$01] Christoph Benzmüller, Andreas Meier, Erica Melis, Martin Pollet, Jörg Siekmann, and Volker Sorge. Proof planning: A fresh start? In Manfred Kerber, editor, *Proceedings of the IJCAR 2001 Workshop: Future Directions in Automated Reasoning*, number DII 06/01, pages 25–37, Siena, Italy, 2001.

[BS98] Christoph Benzmüller and Volker Sorge. A blackboard architecture for guiding interactive proofs. In Fausto Giunchiglia, editor, *Proceedings of 8th International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA'98)*, number 1480 in LNAI, pages 102–114, Sozopol, Bulgaria, 1998. Springer.

[BS99a] Christoph Benzmüller and Volker Sorge. Critical Agents Supporting Interactive Theorem Proving. Seki Report SR-99-02, Computer Science Department, Universität des Saarlandes, 1999.

[BS99b] Christoph Benzmüller and Volker Sorge. Critical Agents Supporting Interactive Theorem Proving. In P. Barahona and J. J. Alferes, editors, *Progress in Artificial Intelligence, Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA-99)*, volume 1695 of *LNAI*, pages 208–221, Évora, Portugal, 21–24, September 1999. Springer Verlag, Berlin, Germany.

[BS00] Christoph Benzmüller and Volker Sorge. ΩANTS – an open approach at combining interactive and automated theorem proving. In M. Kerber and M. Kohlhase, editors, *Proceedings of the Calculemus Symposium 2000*, St. Andrews, United Kingdom, 6–7 August 2000. AK Peters, New York, NY, USA. forthcoming.

[Bun88] Alan Bundy. The use of explicit plans to guide inductive proofs. In *Conference on Automated Deduction*, pages 111–120, 1988.

[Bun02] Alan Bundy. A critique on proof planning. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, Lecture Notes in Computer Science. Springer, 2002.

[CAB$^+$86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986.

[CS00] Lassaad Cheikhrouhou and Volker Sorge. PDS — A Three-Dimensional Data Structure for Proof Plans. In *Proceedings of the International Conference on Artificial and Computational Intelligence for Decision, Control and Automation in Engineering and Industrial Applications (ACIDCA'2000)*, Monastir, Tunisia, 22–24 March 2000.

[EM99] Jörg Siekmann Erica Melis. Knowledge-based proof planning. *Artificial Intelligence*, 1999.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, 2nd edition, 1996.

[Gal86] Jean H. Gallier. *Logic for Computer Science. Foundations of Automatic Theorem Proving.* Harper and Row Computer Science and Technology Series. Harper and Row, Cambridge;Philadelphia;San Francisco, 1986.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen II. *Mathematische Zeitschrift*, 39:572–595, 1935.

[GHL60] H. Gelernter, J. R. Hansen, and Donald W. Loveland. Empirical explorations of the geometry theorem machine. In *Proceedings Western JCC 17*, 1960.

[GMW79] Michael J. Gordon, Arthur J. Milner, and Christopher Wadsworth. *Edinburgh LCF.* LNCS. Springer, 1979.

[Hua94] Xiaorong Huang. Reconstructing proofs at the assertion level. In Alan Bundy, editor, *Proc. 12th Conference on Automated Deduction*, pages 738–752. Springer-Verlag, 1994.

[KKS98] Manfred Kerber, Michael Kohlhase, and Volker Sorge. Integrating Computer Algebra Into Proof Planning. *Journal of Automated Reasoning*, 21(3):327–355, 1998.

[McC90] William McCune. Otter 2.0. In Mark Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction (CADE–10)*, volume 449 of *LNAI*, pages 663–664, Kaiserslautern, Germany, 1990.

[McC94] William McCune. A Davis-Putnam Program and Its Application to Finite First-Order Model Search: Quasigroup Existence Problems. Technical Memorandum ANL/MCS-TM-194, Argonne National Laboratory, USA, 1994.

[McC97] William McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[Mei00] Andreas Meier. Tramp: Transformation of Machine-Found Proofs into ND-Proofs at the Assertion Level. In David McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE–17)*, volume 1831 of *LNAI*, pages 460–464, Pittsburgh, PA, USA, June 17–20 2000. Springer Verlag, Berlin, Germany.

[Mei03] Andreas Meier. *Proof-Planning with multiple strategies.* PhD thesis, University of the Saarland, 2003. forthcoming.

[Mel98] Erica Melis. AI-techniques in proof planning. In *European Conference on Artificial Intelligence*, pages 494–498, 1998.

[Mil85] R. Milner. The use of machines to assist in rigorous proof. In C.A.R. Hoare and J.C. Sheperdson, editors, *Mathematical Logic and Programming Languages*, pages 77–87. Prentice-Hall, 1985.

[Pau86] Larry Paulson. Natural deduction as higher order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Pau94] Lawrence C. Paulson. *Isabelle; A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer, Berlin;Heidelberg;New York, 1994.

[PB02] Florina Piroi and Bruno Buchberger. Focus windows: A new technique for proof presentation. In Jaques Calmet, Belaid Benhamou, Olga Caprotti, Laurent Henocque, and Volker Sorge, editors, *Artificial Intelligence, Automated Reasoning and Symbolic Computation*, number 2385 in LNAI, pages 337–341. Springer, 2002.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[RS93] Peter J. Robinson and John Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal of Logic Computation*, 3(1):47–61, 1993.

[RS01] Julian Richardson and Alan Smaill. Continuations of proof strategies. In Tobias Nipkov Rajeev Gor, Alexander Leitsch, editor, *Proceedings of International Joint Conference on Automated Reasoning*, pages 130–139, 2001.

[Sch77] K. Schütte. *Proof Theory.* Springer Verlag, 1977.

[Smu68] Raymond R. Smullyan. *First Order Logic.* Springer, 1968.

[Sor00] Volker Sorge. Non-Trivial Symbolic Computations in Proof Planning. In Hélène Kirchner and Christophe Ringeissen, editors, *Proceedings of Third International Workshop Frontiers of Combinning Systems (FROCOS 2000)*, volume 1794 of *LNCS*, pages 121–135, Nancy, France, March 22–24 2000. Springer Verlag, Berlin, Germany.

[Sor01] Volker Sorge. ΩANTS*: A blackboard architecture for the Integration of Reasoning Techniques into Proof Planning.* PhD thesis, University of the Saarland, 2001.

[SRV01] R. Sekar, I.V. Ramakrishnan, and Andrei Voronkov. Term indexing. In Alan Robinson Andrei Voronkov, editor, *Handbook of Automated Reasoning*. Elsevier Science, 2001.

[VBA03] Quoc Bao Vo, Christoph Benzmüller, and Serge Autexier. An approach to assertion application via generalized resolution. SEKI Report SR-03-01, Fachrichtung Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2003.

[Wal90] Lincoln A. Wallen. *Automated Proof Search in Non-Classical Logics. Efficient Matrix Proof Methods for Modal and Intuitionistic Logics.* MIT Press, Cambridge, Massachusetts;London, England, 1990.