General-Purpose Reasoning Assistant System

南, 俊朗
Graduate School of Information Science and Electrical Engineering, Kyushu University

https://doi.org/10.11501/3150887

出版情報：九州大学, 1998, 博士（理学）, 課程博士
バージョン：
権利関係：

General-Purpose Reasoning Assistant System

南　　俊　朗

①

General-Purpose Reasoning Assistant System

Toshiro Minami

February 1999

Department of Informatics
Kyushu University

Abstract

In these days, logics are playing an important role not only in Philosophy and Mathematics but also in Computer Science, Artificial Intelligence, and in other areas such as Aesthetics, which used to be considered in the opposite position to logic. In these areas many kinds of logical systems have been proposed and studied for coping with a wide variety of problem domains. In order to deal with these logics, various types of reasoning systems have been developed. Automated theorem provers have been investigated to pursue the mechanisms to prove the practical theorems automatically. Proof checkers aim to help the users with creating rigorous proofs by providing the means of checking the correctness of the proofs given by them. Proof constructors aim to provide proof editing environments in which the users interactively construct various proofs.

However most of these systems were developed for a special purpose. Their underlying logics are fixed in advance and the facilities provided by them are specific to those logics. Therefore it is difficult for them to cope with a wide variety of logical systems in a uniform framework. Another problem is that it is also difficult for such systems to use in formulating a new logical system for handling a new problem domain.

This thesis aims at establishing a system architecture that overcomes these problems. In order to achieve this aim, we start with modeling the reasoning process of humans, and clarify the issues to be investigated in the system. Based on this model, we establish an architecture of general-purpose reasoning assistant system called EUODHILOS, which provides the users with the facilities for (i) defining their own logical systems in such a way that they can specify the systems in a natural representation, and (ii) constructing proofs of theorems of the defined logical systems in a flexible reasoning assisting environment.

Specifically, a logical system in EUODHILOS consists of a language system and a derivation system. A language system is specified with a new symbol definition and a syntax description of the logical expressions so that the user can use his or her own representations of symbols, terms, and formulas. A derivation system is specified in the natural deduction style, where the user is free to choose his or her own style of formulation for the system that consists of any combination of axioms, inference rules, and rewriting rules. These constituents are naturally represented in tree-form so that the user can define, recognize, and modify them easily. The proof

assisting environment provides some flexible and easy-to-use facilities for the user with tree-form proof representation to easily recognize the proof structure, free choice of derivation directions, rich proof editing functions such as connection and separation, and schematic reasoning with metavariables.

The feasibility and generality of the architecture is verified by developing two systems on different platforms. The usefulness of this architecture is demonstrated also with an application example to knowledge acquisition support systems in the network environment.

Through the implementations and experimentations presented in this thesis, we demonstrate the potential and importance of the approach of the general-purpose reasoning assistant system like EUODHILOS to a logic-based interactive problem solving in a wide variety of domains.

Acknowledgments

First of all, I would like to express my sincere gratitude to my thesis supervisor, Setsuo Arikawa, for his continuous warm encouragement and valuable comments and helpful advice. I am also grateful to my thesis committee members Fumihiro Matsuo and Sachio Hirokawa for valuable comments and discussions. Especially Sachio Hirokawa has been giving me warm encouragement, thoughtful comments and stimulating discussions. I would like to express my sincere thanks for everything they have done for me.

I am deeply indebted to Hajime Sawamura and Takeshi Ohtani for their collaboration on the topic of this thesis. It has been quite valuable, fruitful and exciting for me to discuss and exchange ideas with them. The research in this thesis will never be accomplished without their contribution. I am also indebted to Kaoru Yokota and Kyoko Ohashi for their participation to our project with discussions, system design and development.

I greatly appreciate the helps from Bob Meyer, Michael McRobbie, John Slaney, and other members of the CISR of Australian National University during, before and after my staying with them. It was a valuable and stimulating experience to be with and to cooperate with them. I would also like to thank them for giving me a good research environment and time for deepening my research. It was also valuable for me to have discussions with Peter Eades, Norman Foo, Boon Toh Low, Tao Lin, and many other people in the Australian academia.

Special thanks are due to Shigeru Sato, Hirobumi Takanashi, Jyunichi Tanahashi, and Kazuo Asakawa for their supports and encouragement. Thanks to them, I had such a precious and unforgettable experience of working and studying as a company employee and a student at the same time, which was a hard, but enjoyable one indeed.

Many thanks are also due to my colleagues: Jun Arima, Mitsuru Oda, Takeshi Ohtani and many other people in Fujitsu Laboratories Limited including those people who used to work with me in ISIS-SIS. I thank again to Jun Arima, and to Seishi Okamoto, for helping me with preparing this thesis by giving me valuable and informative advice.

Part of the work in this thesis was carried out as a part of the major research and development of FGCS project conducted under the program set up by MITI. I would like to thank to the people of ICOT who supported our work including Kazuhiro Fuchi, Koichi Furukawa and Ryuzo Hasegawa.

I would like to give my special thanks to my elder brother Tetsuhiro Minami, who is always willing to support me and has given me continuous encouragement and valuable advice.

Finally, my deepest thanks are due to my parents and my family. My mother, Fujie, and my late father, Fumio, have been supporting and encouraging me with pride and affection in my whole life. My wife, Tomoko, and my daughters, Mariko and Emiko, have been with me and given me supports and energies to working during the period of the research in this thesis.

Contents

List of Figures

Chapter 1

Introduction

Every universe of discourse has its logical structure.
by S. K. Langer[50]

Reasoning is an essential and prominent ability of human beings. The history of logical formalization goes back to Aristotle who investigated the patterns of reasoning and found some rules called syllogisms. From those days logics have been investigated by philosophers in order to clarify the rules of reasoning and thinking mechanisms of humans. In the 19th century Boole and Frege made the foundation of formal, or mathematical, logic theory which we use now. Since then a lot of mathematicians have been involved in the research of formal logics.

The fundamental mechanism of digital computers is theoretically originated in Boolean algebra, which is created in the research for formalizing the rules of reasoning. Based on such a relationship, quite a lot of automated theorem provers[11] have been investigated and developed from the early days of computer history. An automated theorem prover solves a problem that is represented in a logical formula by a mechanized proving algorithm. The research field called computational logic is originated from such research activities.

Various researches on computational logic and developments of a variety of computer systems that deal with logics have been performed so far. For example, from the research on automated theorem proving, Boyer-Moore theorem prover[7], Otter[56], MGTP[34] were developed. FINDER[107] solves finite domain problems, which was originated from the research for generating finite models of logical systems[106]. One of the main roots of the research on proof constructors is AUTOMATH[8]. LCF[27], NUPRL[15], Isabelle[85] and many other systems inherited the philosophy and methodology of this system and gives improvements to it. The purpose of research

and development of such reasoning assisting systems lies basically in strengthening the reasoning power of humans so that human reasoning proceeds more efficiently and more rigorously than without them. The rapid development and increasing of popularity of computers make such attempts of assisting of human reasoning much easier to achieve than before.

General-purpose reasoning assistant system(G-RAS) is an approach to deal with a variety of logical or formal structures under the support by computers. The largest motivation for investigating such system concept is that quite a number of logical systems appear these days. They are created, investigated, used, and are playing an important role not only in mathematics and philosophy but also in other fields such as computer science, cognitive science, and artificial intelligence[40; 110; 113] for modeling computation mechanisms, and for representing and manipulating knowledge. Furthermore, surprisingly enough, it is also applicable and useful in aesthetics which has been thought of as being in an opposite position to logic[48; 50], as well as in other scientific theories[4; 83; 116]. Specifically, it can be said that logics provide expressive devices for objects and their properties, and inference capabilities for reasoning about them. It is also the case that symbol manipulating methods provided in logics are basically common to all scientific activities.

Much work has been devoted to special-purpose reasoning assistant systems whose underlying logics are fixed[9; 15; 27; 43; 111; 114]. However a lot of tedious work must be done to implement such systems in order to deal with a number of logics. Not only must the parser for reading commands and expressions be developed, but also the pretty printer program, or unparser, for displaying the results, the facilities for defining the derivation system, the proof manipulation functions, and many other programs must be developed for each of the logics under construction. Thus, this approach is not a relevant one for developing systems that deal with many target logics, especially for dealing with them in a uniform framework. To overcome this problem, G-RAS system provides a means of defining logics and gives a proof construction supporting facilities for the defined logic. In this approach, the user can easily define and modify the definitions of the target logics until it is satisfactory.

Another problem of the special-purpose systems is that they cannot cope with the attempts of formulating new formal systems from scratch. As has been pointed out, a large number of logical or formal systems have been and are going to be developed, and thus there is a big need for the computer systems that assist the formalizations of many problem domains. Such systems are supposed to give assistants to the process of describing, experimenting, and refining the logical systems that is performed

essentially by trial and error. A special-purpose reasoning assistant system is not good for such attempts to use because the only way to deal with a logical system in such a system is by converting the expressions and rules of the problem domains into the underlying logic of the system and the user is supposed to handle the expressions through the converted, and thus indirect, expressions. It is also a problem that, in most of the cases, only a small variety of logical systems can be converted into the underlying logic. It might be worth for it if we do such conversions in an automated theorem prover, because the conversion is only one way. That is, the conversion is done only from the expressions in the problem logic into the corresponding ones in the underlying logic of the system. In interactive systems, the user should also convert the expressions in the system's underlying logic to the expressions in the logic of problem domain. This direction of conversion is much harder than the other one, thus it is virtually impossible for the user to perform the reasoning in such an interactive environment. To conclude, we can say that it is impossible to take the approach of the special-purpose systems for developing new logical systems that are expected to be in a wide variety of formulation styles. Therefore taking the approach of the general-purpose systems is the only choice for the user in these attempts.

In order to establish general-purpose reasoning assistant systems, we have to pursue a couple of subjects. The first one is the "generality" of the system. As S. K. Langer pointed out, we recognize that "Every universe of discourse has its logical structure.[50]" That is, a thought that for each object which we mention to talk with, there must be a logic best suited for expressing about it. For example, linear logic[26] is the logic for dealing with resources and it is also used to represent parallel processing[31]. Modal logics[37] deal with modalities of propositions and are used in hardware and software verifications, knowledge representation, and non-monotonic reasoning. In order to assist human reasoning for such objects, the G-RAS system is required to have the power to describe a wide variety of the existing logical structures and deal with the expressions under these logics.

The second important subject to be pursued for G-RAS is usability. Considering the generality of G-RAS, it is almost impossible to prepare an efficient theorem-proving, or decision, procedure in advance for such a wide spectrum of formal systems. So the basic reasoning steps should be controlled by the user. From this observation a G-RAS system should be designed to be an interactive system, thus the usability is an important matter.

Lastly the study of G-RAS systems in the network environment is also important these days. In these several years the computer network, specifically Internet,

is coming to be popular quite rapidly and a lot of computers are connected each other. In such an environment of computerized societies huge amount of information are stored in computers and we can access them easily through the network like Internet. Because we are in such a situation, the technologies that help with creating knowledge are getting more and more attention and extracting knowledge from data storage and having good use of it is going to be a crucial technology. The reasoning systems that deal with knowledge is getting a new light under such a circumstance and logic-based knowledge extracting and manipulating systems, that include reasoning assistant systems, are one of the core components of such knowledge management systems.

The aim of this thesis is to investigate an architecture for general-purpose reasoning assistant systems that provides the users with interactive and easy-to-use reasoning environment for a variety of formal systems. In order to achieve this aim, we start with modeling the reasoning process of humans, and clarify the issues to be investigated in the system. Based on this model, we establish an architecture of general-purpose reasoning assistant system called EUODHILOS[1], which provides the users with the facilities for (i) defining their own logical systems in such a way that they can specify the systems in a natural representation, and (ii) constructing proofs of theorems of the defined logical systems in a flexible reasoning assisting environment. Specifically, a logical system in EUODHILOS consists of a language system and a derivation system. A language system is specified with a new symbol definition and a syntax description of the logical expressions so that the user can use his or her own representations of symbols, terms, and formulas. A derivation system is specified in the natural deduction style, where the user is free to choose his or her own style of formulation for the system that consists of any combination of axioms, inference rules, and rewriting rules. These constituents are naturally represented in tree-form so that the user can define, recognize, and modify them easily. The proof assisting environment provides some flexible and easy-to-use facilities for the user with tree-form proof representation to easily recognize the proof structure, free choice of derivation directions, rich proof editing functions such as connection and separation, and schematic reasoning with metavariables. The feasibility and generality of the architecture is verified by developing two systems on different platforms. The usefulness of this architecture is demonstrated also with an application example to knowledge acquisition support systems in the network environment. We demonstrate the potential,usefulness, and importance of the approach of the general-purpose reasoning assistant system like EUODHILOS to a logic-based interactive problem solving in a wide variety of domains, through these implementations and experimentations.

[1] "EUODHILOS" is an acronym of the statement that "Every universe of discourse has its logical structure." by Langer[50], and reads "you-oh-dee-los."

The rest of this thesis is organized as follows. In Chapter 2, we characterize the concept of general-purpose reasoning assistant system(G-RAS) through comparing it with other types of reasoning assisting systems. An automated theorem prover(ATP) is different from G-RAS in the sense that the user gives the goal formulas to an ATP and it tries to automatically find a proof of the given goal formulas. In a G-RAS, on the other hand, the user takes the leadership of the proving process, where the system's role is to assist the user's own reasoning. In a proof checker(PC), the user creates his or her own proofs and gives them to the system which are described in the proof description framework provided by the system. The system then checks the given proofs if they are valid or not. In G-RAS, the proof construction is performed cooperatively with the user and the system. Proofs are checked at the same time as they are constructed so that invalid proofs would never be created. A proof constructor, or a proof editor(PE), resembles to G-RAS in this respect. Their differences are on that the underlying logical system is fixed in a PE, whereas G-RAS can deal with a wide variety of logical systems by allowing the user to define his or her own logical systems in a metalogical description framework.

In Chapter 3, we first consider the process of human reasoning with setting up a model of the process. Then we design a general architecture of a G-RAS system which we call the EUODHILOS architecture. In the architecture a formal system is described in a logical framework, where it consists of the language system and the derivation system. A language system is specified mainly with syntax specification with a facility for defining new symbols. A derivation system is specified in a combination of axioms, inference rules, and rewriting rules. Such a framework is easy-to-describe, easy-to-recognize, and expressive enough for specifying a wide variety of formal systems. Two implementations have been accomplished on different platforms based-on this general architecture. EUODHILOS-I[66; 67; 97; 104] was implemented on the PSI(Personal Sequential Inference) machine with SIMPOS operating system. EUODHILOS-II[68; 79; 80; 81] was implemented on top of GNU Emacs/Epoch/Mule systems. Despite their conceptual system models are common, their details for specific functions are different, which reflects the difference of platforms and specific designing policies. The principal designing policy on interface for the former system is to make good use of the SIMPOS window system. Thus it became the GUI-based system where the basic operations are performed with the

mouse, icons, menus, and so on. In the latter system, because Emacs is well known in its keyboard oriented usable environment, most well-used operations are assigned to key sequence(s) so that they are performed with keyboard and the requirements to mouse operations are minimum.

In Chapter 4, we are concerned with the description framework for language systems. By using the expressions for the objects and concepts in the target domain of the user's own choice, he or she will be easier to capture the situation and thus easier to manage the reasoning process than to do these things without such a feature. A language system consists of symbol definition facility on which users can define and use their own logical symbols, and the syntax description facility on which users can specify what expressions are used as terms, formulas, judgments, and other logical expressions. However the specific methods of specifying the language systems are different between EUODHILOS-I and EUODHILOS-II systems. In the former system, the new symbols are defined in a built-in font editor of the operating system and they are assigned on a software keyboard and can be typed through it. The syntax is given in the DCGo notation, which is an extension to DCG(Definite Clause Grammar) notation augmented with the constructor declaration. A bottom-up parser and an unparser will be generated automatically when the syntax definition is finished. In the latter system, the new symbols are assigned with the X-window system architecture and they are typed through the key sequences bound to them or with the character codes. The syntax is based on the BNF[36] notation with some additional information such as root formula specification for top-down parsing and metavariable representations. The system generates data for the fixed parser and unparser.

In Chapter 5, we are concerned with the framework for specifying the derivation systems of EUODHILOS. A derivation system is a description of the structure of the target domain in a logic-based framework. In our framework, a derivation system consists of any combination of axioms and two types of derivation rules; the inference rules and rewriting rules. These rules are specified in the natural deduction style and are represented in tree-form. This specification framework is rich enough to cover quite a wide variety of formulation styles such as Hilbert style, sequent style, natural deduction, and even other styles that will not be used in ordinary logical formulations. In Hilbert style formulations, information about the structure is given in a set of axioms and inference rules are used in minimum. In sequent style, the primitive expression consists of a sequence of formulas, followed by a symbol called turnstile followed by a sequence of formulas. In a typically formulated sequent calculuses, the derivation systems are given with one axiom in the form $A \vdash A$ and various inference

rules that consists of left and right introduction rules for logical connectives. Natural deduction allows to use the assumptions in the reasoning and typical systems consist only with inference rules, most of which are introduction and elimination rules for logical connectives. The nature of assumption-based reasoning that it is intuitive and easy to use in reasoning is another reason why we have chosen the natural deduction style formalism for derivation system description framework.

In Chapter 6, we are concerned with the facilities for assisting proof constructions, which are easy to use and flexible enough to adapt to a wide variety of reasoning styles of humans. In order to build-up such facilities, the proof assisting environment, which we often call a sheet of thought, provides the features in such a way that partially constructed proofs, or proof fragments, are represented in tree form as in derivation rules, so that users are easy to recognize the organization of the proof fragments. It also enables the users to freely set the derivation direction of the rule applications, to manipulate proofs in various editing functions like connection, separation, instantiation, and some others. Some representation and operation details are different from EUODHILOS-I and EUODHILOS-II. In EUODHILOS-I, proofs are represented in full tree form, which does not omit any part, whereas in EUODHILOS-II, proofs are usually represented in an abridged form where the assumptions, premises, conclusions, and the rule that deduces the conclusion are specified. A full proof is represented in a line form for the full tree representation and a pretty-printing form is obtainable in LaTeX representation. Proof tactics and tacticals are also important in reasoning in order to semi-automate the reasoning steps for straightforward reasoning. We also deal with the description method and internal processing of the application of tactics.

In Chapter 7, we demonstrate the generality and potential usefulness of EUODHILOS systems by experimenting with several examples and show how these logics are actually specified and what sort of proofs have been actually developed. The presented logical systems cover a wide variety of formulation styles from the ordinary classical logic to other systems like modal logic, Hoare logic, linear logic, combinatory logic, and category theory.

In Chapter 8, we present another application example of EUODHILOS to the knowledge acquisition support systems. This example suggests new application possibilities of EUODHILOS systems into the knowledge management field in network environment. One of the notable points in this chapter is that it is important to discriminate the domain-knowledge and meta-knowledge. The domain-knowledge is the data which we use for defining formal or logical systems. This type of knowledge is extracted from the human brain. Meta-knowledge is the knowledge obtained in the

reasoning procedure, such as theorems, derived rules, and tactics. The most promising feature of this comes from that the meta-knowledge is "safe", that is, it does not relate to the validity of the resulting proofs so that they can be circulated and are applicable to various situations without worry.

In Chapter 9, we shed light on the characteristic features of EUODHILOS systems by comparing them with some of the related reasoning systems. As we have mentioned, EUODHILOS systems are designed based on the human reasoning process model so that they are in good balance in the proving abilities and usability of the users.

And finally in Chapter 10, we conclude this thesis and suggest some of the possible future research directions to be investigated.

Chapter 2

Concept of General-Purpose Reasoning Assistant System

The computer systems which are called the computer assisted reasoning systems, or sometimes called simply as theorem provers, are the computer systems that deal with formal systems that are based on the logical formulation and involve in theorem proving in these logics. Various types of reasoning systems have been researched and developed so far. Some systems focus on proving theorems automatically while others intend to efficiently construct a number of proofs in the interaction between human users and the systems. The aim of this chapter is to clarify the concept of general-purpose reasoning assistant system in comparison with other types of such reasoning, or proof construction, systems.

We classify such reasoning systems into the following four types in terms of the aims of the systems and the styles of interaction with the users.

(i) Automated Theorem Prover(ATP)

(ii) Proof Checker(PC)

(iii) Proof Constructor/Editor(PE)

(iv) General-Purpose Reasoning Assistant System(G-RAS)

In the rest of this chapter we take out one by one of the four system types and describe its basic concept by illustrating its typical style of interaction with its users, and discuss its differences from others.

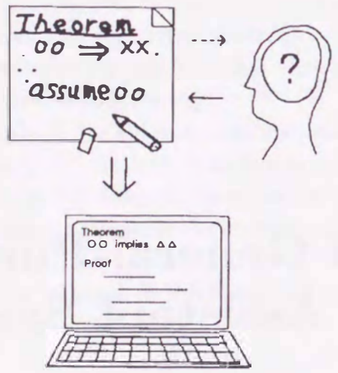


Figure 2.1: Automated Theorem Prover(ATP)

(i) Automated Theorem Prover

An automated theorem prover(ATP) is a system that searches for a proof of the given formula completely automatically. The concept is illustrated by presenting a typical style of ATP in Figure 2.1. In the upper half of the figure, a user tries to prove a formula, which might be his or her conjecture that is supposed to hold in the theory under consideration. The user starts with putting an assumption of the formula. Then he or she gets stuck; having no ideas how to go further. In such a situation the user will use an ATP for help. The ATP system gets the conjecture formula, tries to find a proof of it, and displays a proof of the formula on the screen if it succeeds. The user may get the message that the given formula is unprovable or that the system has failed to prove the formula.

However the scenario is rather too optimistic. Considering the current state of the art of the research for proof-finding algorithms and also theoretical difficulty of automatic theorem proving, there is only a little hope of finding practically useful proofs that should be a very complicated ones in an automatic way by computers and use such systems in practical applications.

From this observation, one of the most suitable ways of applying ATP systems is to use them for finding small and simple gaps included in a large proof. If the gaps of proofs are small enough, the prover will find the proofs effectively.

Many theorem provers are able to save the resulting proofs and register the theorems[7], i.e. the conclusion of the proofs, in the theorem database. The theorems in the database will be used automatically in the proving attempts that follows. By using this mechanism the user can construct a complicated proof with an ATP system "interactively" by giving formulas in a sequence so that the system proves each formula, save it as a theorem, and eventually reaches the final goal formula that is estimated to be very difficult to prove. Another recommendable way of using ATP system is to use it as a component of other types of theorem proving systems for the facility that automatically fills the small gaps in a big proof.

The most characteristic difference of ATP systems from other types of systems is that ATP systems intend to prove the given goal formulas completely automatically. In other words, they are interested in investigating what algorithms and architectures are effective in proving automatically and what mechanisms simulate those of human reasoning.

Comparing human reasoning and computer reasoning of ATP systems, the former one is good at making a general plan how to find a proof and the latter is good at doing things fast and with accuracy. Thus the ideal collaborative style of humans and computers in reasoning is that the human user gives a rough sketch of proofs and the computer gives actual proofs by following to the proof plan. If some gaps are too big for computers to prove or disprove the user gives advice how to do with them.

Because the automated theorem proving has been a kind of central research interest for a long time, a lot of researches have been performed and various theorem provers were developed(e.g. BMTP/NQTHM[7], Otter[56], FINDER[107], and MGTP[34]).

(ii) Proof Checker

A proof checker(PC) is a system that checks the correctness of a proof given by the user. As in the opposite view of the ATP, the most important characteristic feature of PC systems is that the proofs are basically created by humans. The role of the computers is to check the given proof and to give a justification to the proof. Figure 2.2 is an illustration of how a proof checker is used. In the upper half of the figure, the user makes a proof of a theorem. A human proof may contain some mistakes including the gaps between lines of the proof which are impossible to fill. In the lower half of the figure, the user gives his or her proof to a PC system. The system provides a proof description framework or language. By using this language,

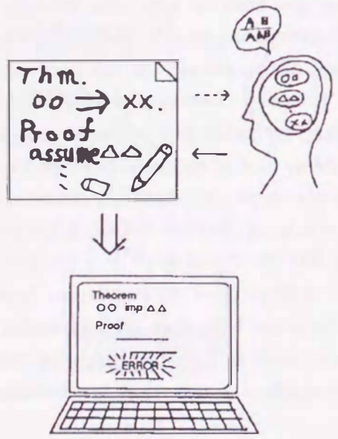


Figure 2.2: Proof Checker(PC)

the user writes the proof and gives it to the system. Then the system checks if the given proof is valid or not. The system may call an ATP for filling small gaps in the proof. If the system finds some errors in the proof, it will show them to the user.

In this system style, the proof description language is the most important subject to pursue, because it should be not only easy to describe but also be able to provide a means for describing a wide variety of proofs, which are supposed to follow the natural reasoning structure of humans. The user would feel bored if he or she has to describe a proof in too detail. On the contrary, if the the gaps between the lines in the description is too big for the proof checking algorithm that runs in the background of the system, the proof will not be verified. In such a case, the user is supposed to refine his or her current proof description and tries again to check the validity of the revised description.

If a user has a proof and wants to verify its correctness, a proof checker might be the best choice for him or her. However if the user begins to find a proof for some formula from scratch, proof constructors(PEs) and general-purpose reasoning assistant(G-RAS) systems might be better choices for the user.

A number of proof checkers have been developed so far. AUTOMATH[8; 9] is the proof checker which has a general description framework in which the user can

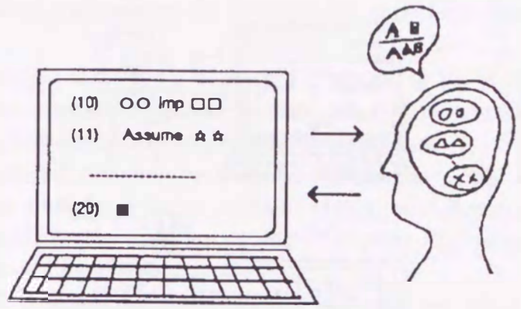


Figure 2.3: Proof Constructor/Editor(PE)

specify how the proofs are constructed. This system is the origin for many systems. PL/CV2[14] is used for proving the correctness of PL/I-like programs. CAP-LA[38; 94] deals with the proofs on linear algebra. MIZAR[111] gives a general framework for describing practical mathematical proofs and publishes the results collected from all over the world.

(iii) Proof Constructor/Editor

A proof constructor or a editor(PE), is a system that supports the user to construct proofs as well as theorems through the interaction between the user and the system. Users manipulate proofs, or proof fragments, by inputting, deleting, applying rules, and combining them. A proof constructor is an editor of proofs in this respect. The function of the proof constructor is included in a G-RAS system as an aid to proof constructions.

Figure 2.3 is an illustration of how the proof constructors are used. The user uses a proof constructor by inputting some commands and tries to build up a proof of a formula which is supposed to be a theorem. Since the proof fragments are verified as they are constructed, the theorems are obtained as their whole proving processes terminate. Thus the features for supporting proof editing, or manipulation, is quite important for proof constructors.

Many proof constructors have been developed so far. For example, LCF[27], EKL[43], IPE[93] and Nuprl[15] are examples of proof constructors.

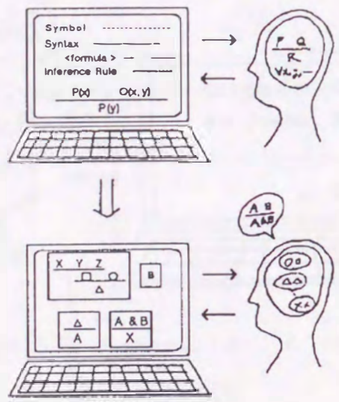


Figure 2.4: General-Purpose Reasoning Assistant System(G-RAS)

(iv) General-Purpose Reasoning Assistant System

A general-purpose reasoning assistant system(G-RAS) is facilitated with a function for defining logical systems in a uniform way in order to deal with a variety of logical or formal structures. It also provides a comfortable and easy-to-use interactive proof construction environment so that the users can proceeds reasoning efficiently. Such a system concept is important according to the philosophy of Langer[50], and to the recognition that the (mathematical) reasoning proceeds through "Proofs and Refutations"[49].

Figure 2.4 is an illustration of how a general-purpose reasoning assistant system is used. In the upper half of the figure which corresponds to the feature of generality, the user specifies each of the components of logic, i.e., symbols, syntax of expressions including logical formulas, inference rules, etc. In the lower half of Figure 2.4, corresponding to the feature of proof constructing environment, the user tries to construct proofs of theorems under the logic defined in the previous step. Proofs will be constructed by trial and error.

The G-RAS system is different from ATP system in the sense that proofs are searched and constructed in the commands of human users, whereas in ATP it is the system that finds the proofs. As has been discussed, the ATP system can well

participate in the G-RAS system for filling small gaps between assumptions and conclusions.

The G-RAS system is different from the PC system in the sense that the proofs are constructed in a cooperation of the user and the system, whereas in PC, it is the user's role to create the proofs to be checked by the system.

The most significant difference between the PE and the G-RAS systems is that in the former systems, underlying logics are fixed, while in the latter systems, underlying logics can be defined by the user. From this standpoint PE systems may be called special-purpose reasoning assistant systems.

The special-purpose reasoning systems give a natural and efficient method of developing systems as long as the target domain is fixed and, thus, its basic logical structure is fixed. Its best advantage is that it is quite easy to augment the system by adding specific facilities that cope with the fixed underlying logics.

However, as we have already found and recognized that in these days the logical methodology becomes a kind of paradigm of promoting mathematics, computer science, artificial intelligence, cognitive science, and so on, we have to suppose that a wide variety of formal structures that are waiting to be dealt with. If a new formal system is similar to an old one in its structural character, it will not be a hard work to modify the old one and adjust it to the new structure. However, as we are supposing to deal with a wide spectrum of logical structures, implementing an interactive system for developing proofs is a discouraging and laborious task for any style of presentation of these logics. For example, one must implement a parser, term and formula manipulation operations (such as substitution, replacement, juxtaposition, etc.), inference rules, rewriting rules, proofs, proof strategies, definitions and so on, depending on each logic under consideration.

Thus, it is desirable to find a general and uniform theory of logics and implement a general-purpose reasoning assistant system that captures the uniformity and idiosyncrasies of a large class of logics so that much of this effort can be expended once and for all. It will be worth to note that a similar observation and motivation can be found in the papers of [30] and [32], although the approaches differ. This was our first motivation for pursuing the G-RAS system. The second motivation for G-RAS comes from a rigorous approach to program construction. Abrial[1] claims that a general-purpose proof checker could be perhaps one of a set of tools for computer aided programming when we consider program construction from various theories. We are certainly in a situation that before embarking on the construction of a program we need to study its underlying theory, that is, to give a number of definitions,

axioms and theorems which are relevant to the problem at hand. Note that every program, or universe, to be constructed, or studied, has its underlying theory, or logical structure.

All this discussion may be summarized as, to borrow Langer's statement[50], "Every universe of discourse has its logical structure". Thus it eventually supports our discussions about the need and significance of the generality in reasoning assistant system from the philosophical point of view.

Chapter 3

Design and Implementation

The aim of this chapter is to present an architecture that realizes the concept of general-purpose reasoning assistant system. In order to achieve this aim we take the approach that firstly, we establish a conceptual model of human reasoning process for formal reasoning. By taking this approach we are able to have a clear view on G-RAS system and also able to understand the role and importance of features of the implemented system from the viewpoint based on the underlying process model. Then we establish a generic organization for such systems, which we call the EUODHILOS architecture. In this architecture a G-RAS system consists of two major components; the component for specifying logic, or formal system, and the component for proof construction, which we often call the "sheet of thought". A target formal system, i.e. a logic, consists of the specifications of the language system and the derivation system by following to the methodology of logical system formulation. On a sheet of thought, proofs are constructed based on the natural-deduction style, where users may put not only axioms and theorems but also assumptions and goals as the starting proof fragments. Proof fragments will be manipulated through rule applications, proof connections and separations, instantiations to metavariables and other operations, and eventually the user will get the complete proofs of goal formulas and other theorems in the target formal system. The actual EUODHILOS systems will be implemented by instantiating the actual representations and actual operation methods on this generic architecture. By taking such an approach, the implemented systems following this architecture have the common conceptual operation model even if the actual user-interfacing facilities are different. In this thesis we will demonstrate this assertion by describing and discussing the common and different features of two implementations on different platforms, which realizes the EUODHILOS architecture.

This chapter is organized as follows.

In Section 3.1, we present a model of human reasoning process, which consists of three phases; observation, formalization, and verification with deduction. We describe each phase and discuss what a G-RAS system can do for such reasoning phases.

In Section 3.2, we propose a generic model of the system called EUODHILOS architecture by considering the reasoning process model and the features of the G-RAS system. The system consists of two parts. The first part is the assisting facility for designing logical systems that are going to be dealt with the system. The other part is the assisting environment for proof construction on which users are trying to find proofs with various reasoning styles.

In Section 3.3, we deal with implementation issues. By describing and discussing about the two implementations, i.e. EUODHILOS-I and EUODHILOS-II systems, we demonstrate how the generic architecture contributes to the common conceptual operational look-and-feel even if their implementation platforms and thus design details are different.

In Section 3.4, we conclude this chapter by summarizing the discussions and observations.

3.1 Human Reasoning Process Model

A logic is, in nature, a formal representation of a recognition of a human for the way of doing things. Therefore it is not surprising that there may exist a number of logics for representing one target domain. In fact it is well known that a logic has various styles in its formulation such as Gentzen's LK, NK, Hilbert's linear style, Fitch style, etc., and that they are mathematically equivalent. From the viewpoint that a formulated system is a representation of how the human recognizes the structure of the target domain, we have to deal with these various logical formulations as different[24]. This understanding about logic-based formal representation is a basis of our approach to the G-RAS system, on which the realization of generality is a major concern. Other philosophical aspects of the generality from a logical point of view can be found in [24] and [50].

In order to investigate a G-RAS architecture for such logical formalizations, we start with setting a model of the process of human reasoning, which is illustrated in Figure 3.1. In the figure, the process consists of four basic steps, which can be divided into the following three major phases.

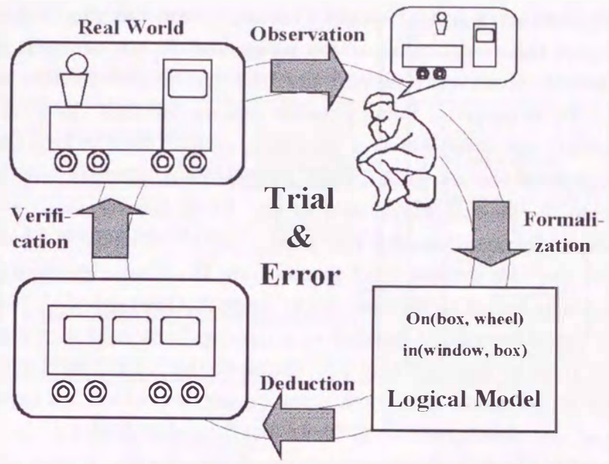


Figure 3.1: The Model of Human Reasoning Process

(1) **Observation:** Forming mental images about the objects or concepts,

(2) **Formalization:** Constructing a logical model that describes the mental images,

(3) **Deduction and Verification:** Examining the models to make sure that they coincide with the real world and mental images.

A formal reasoning process starts with the first phase when one observes the real world and acquires some mental images about objects, relations, and structures in the world, and when one wants to make clear what they are like or wants to solve some of the problems in the domain.

In order to make clear the mental images, one has to formally describe these concepts so that his or her recognition can be shared with other people. In this thesis, a formal framework for describing objects or concepts is called a "logical" framework, and a description which specifies such mental image in a logical framework is called a "logical model". The second phase is the one for constructing such logical, or formal, models. In this phase, the user has to make clear what kinds of objects, concepts, or relations appear in the universe of discourse, or the target domain, and has to decide which of these concepts should be represented in the formulated logical model.

If we can construct a logical model that represents the target domain exactly, we can terminate the model construction phase and do our problem solving in the constructed model. However, it is rare that the first model we have constructed is good enough for our purpose. Some possible reasons are that the model lacks some descriptions about the objects and/or structural properties, the formalized representation does not meet the intention of the human, he or she has misunderstood the problem domain in the real world, and so on. From these reasons we have better to check if the model is sufficiently well built. The third phase is needed for such a purpose, where the user derives some results from the formal model and verifies the results if they really match to the real target domain. One typical checking method is like this. The user knows some number of properties that hold in the target domain and he or she tries to deduce these properties in the logical model at hand. The model appears to be insufficient when some properties, which are expected to hold in the image of the objects, fail to prove in the formulated model. In this case, the user has to modify the logical representations of the objects. It may also occur that the properties that are not supposed to be hold in the real domain hold in the model. If some kinds of mismatches are found the user has to repeat the process from the beginning; observing the target domain in the real world again and trying to find out what are wrong with the current logical model, then re-formalizing the model by modifying the current model, and so on.

It is not conceivable that phase (1) could be aided by a computer system since some part of phase (1) is very creative and deeply depends on the mental activities of humans. On the other hand, it is very likely that phases (2) and (3) could be largely supported by a system by allowing modifications or revisions of the definition of the language used for the modeling and by introducing certain reasoning devices. These are the activities that our system intends to support.

3.2 General System Organization of EUODHILOS

In this section we presents a general system organization called EUODHILOS architecture by considering the reasoning process model that was described in the previous section and also by considering the requirements to the G-RAS system as was discussed in Chapter 2. Roughly speaking the phases of the reasoning process that a G-RAS system could give direct assistance are the formalization step and

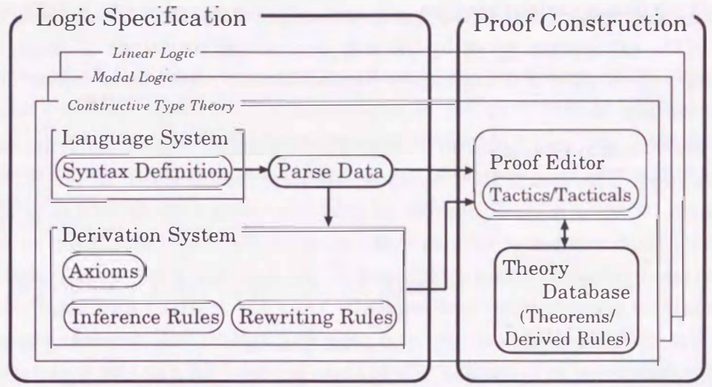


Figure 3.2: General System Organization

deduction step of Figure 3.1.

Two important subjects should be studies in order to assist such steps by a computer system: (1) a general framework for specifying formal systems based on the logical framework which has both general description power and easy to describe, and (2) a flexible supporting environment for proof construction that matches to the various styles of human reasoning. Figure 3.2 illustrates the design of EUODHILOS architecture which is designed to fulfill the requirements to G-RAS systems. A formal system that describes the target domain will be given as a logic specification and the supporting environment for proof constructions will be given by a proof editor, which is often called a sheet of thought.

A logic consists of the specifications of language system and the derivation system. The language system specifies how the logical expressions, i.e., those expressions used in the logic such as formulas, terms, judgments, etc., are constructed from atomic expressions, i.e. the characters. Users can use expressions that fit their preferences by defining an appropriate language system. Take, for example, the quantification expression in first-order logic. Someone might prefer to use the expression "$\forall \mathtt{x}.\phi$" while other ones rather use "$\forall \mathtt{x}:\phi$" or even "$(\mathtt{x})\phi$". This feature naturally helps the user easily capture and understand the meaning of the expressions because they are represented in the user's preferred and thus easy-to-recognize form. In the later activities, i.e. in derivation system specification and in proof construction, the description

of the language system will be used automatically for parsing, i.e. analysis and conversion of the expressions given by the user represented as strings of characters, and unparsing, i.e. creation of string expressions of internal representations for displaying them to the user.

The derivation system consists of three kinds of data: axioms, inference rules, and rewriting rules. The user is free to choose the formulation style. In the Hilbert-style formulation, there is a small number of inference rules such as modus ponens, and relatively a large number of axioms that represent the major structures of the logic. In the sequent calculus formulation, most of the data are given as inference rules in which formulas are represented as sequents. Only the starting formula, for example, "$A \vdash A$", is given as an axiom. A term rewriting system will use only the rewriting rules. It is also possible to combine these three kinds of data in this architecture. For example we can define a derivation system based on the sequent calculus formulation mixed with several rewriting rules. In this way the formulated model is not necessarily based on the ordinary logical systems. Such a freedom of formalization comes from our philosophical standpoint that a logical system represents the formal system that expresses our recognition of the target domain as was discussed in the previous section.

When the logical model is specified it is the time for the user to move to the deduction and verification phase where various theorems are proved, conjectures are verified, and useful derived rules and theorems are established and saved for later use. Proofs are represented in tree-form so that it is easy to capture the overall structure of the proofs. Then the user will have better perspective and he or she is able to make a plan for reasoning with ease and confidence. The proof editing environment, i.e. sheet of thought, supports flexible proof construction styles. For example, both forward derivation (i.e., deriving a new result from known ones) and backward derivation (i.e., deriving one or more subgoals from the given goal) are possible. It is even possible to mix both types of derivation at any time with ease. Connecting small proof fragments and creating a big one should also be supported in this system. Such supporting facilities for proof construction intend to incorporate not only such a flexible proving methodology but also the methodology of science such as Lakatos's mathematical philosophy of science[49] and Kitagawa's relativistic logic of mutual specification[44].

The resultants of the proofs can be saved either as theorems or as derived rules and then used in the later proving processes as if they were primitive axioms or inference rules. These are the quite useful facilities for interactive reasoning systems where users can save the results which, they think, are useful in later reasoning processes.

As the result the proofs becomes more compact and easy to understand by using such results.

Tactics and tacticals will be useful for automating derivations. Users can get the result, which is derived after a certain number of ordinary rule applications, in one application of a tactic. For G-RAS systems this facility is useful to reduce the burden of the users in proving boring proofs such as the straightforward ones but still need a long chain of deductions.

3.3 System Implementation

The actual G-RAS systems will be implemented according to the generic system architecture of EUODHILOS. By taking such an approach we can discriminate the features that come from the system architecture and those come from the features of platform and system designing policy. In our G-RAS system the architecture deals with conceptual organizations of various concepts such as how the whole system is organized, what components a formal system consists of, how proofs are represented and dealt with, and so on. On the other hand, the actual operations for achieving these conceptual operations should be determined according to the implementation language, platform, and designing policy.

In fact, EUODHILOS architecture has two implementations on different platforms. In this section, we first describe each implementation and then we discuss how they are designed by considering the characters of the platforms and designing policies.

(1) EUODHILOS-I: Implementation on PSI

The system which we call EUODHILOS-I[66; 67] was developed in the Fifth Generation Computer Systems Project of Japan, it is implemented in ESP(Extended Selfcontained Prolog) language which is an object-oriented Prolog for PSI(Personal Sequential Inference) machines on SIMPOS operating system. The SIMPOS operating system provides a window environment and our system is designed to exploit the bit-map display with multi-window environment, mouse, icon, pop-up menu, etc. Needless to say, Prolog serves as a good implementation language for theorem provers and interactive reasoning systems since they directly implement search and unification which are essential operations for traversing a search space for a proof and manipulating formulas and proofs. Object-oriented facilities of ESP have played an important role in the implementation as well since it is a kind of generic or metasystem in which

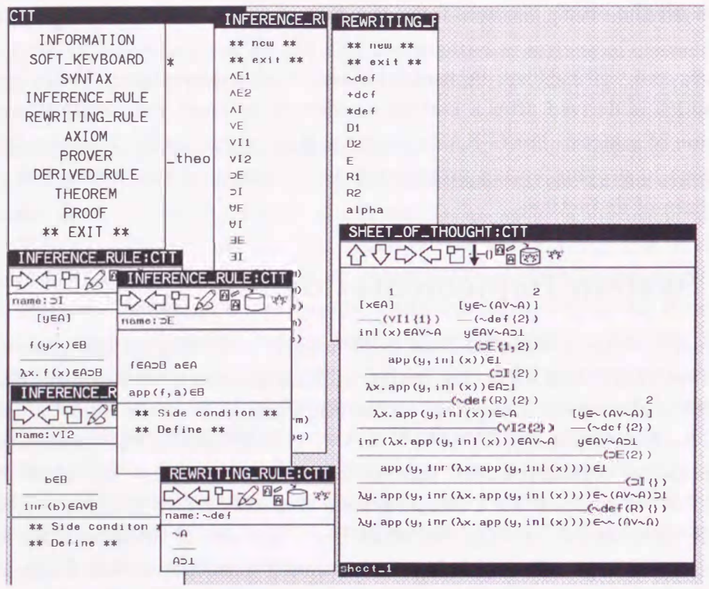


Figure 3.3: Example EUODHILOS-I Screen

each logic is to be constructed as an instance object of the class "logic".

Figure 3.3 is an example screen of this system. At the top-left corner of the screen is a system window for the theory named CTT(Constructive Type Theory), where various functions for dealing with theory data will be invoked from. Actually, behind of this window is a window that displays a list of logics or theories[1] that are already defined in the system. The system window for CTT is invoked from this window. In the upper-center part of the screen are lists of inference and rewriting rules. From these windows we either define a new one or display/update the already defined inference and rewriting rules respectively. Three inference and one rewriting rules are displayed in the lower-left corner of the screen. In the lower-right corner of the screen is an example window of sheet-of-thought, on which proofs are displayed in tree-form.

The operations that are used very often can be invoked from the icons listed in

[1]In this thesis we use the term "logic" for the formal system described in the formulation framework. Thus, the terms "logic" and "theory" are used identically.

the upper line, which represent the functions for, from left to right, scrolling upward, scrolling downward, scrolling to the right, scrolling to the left, resizing the window, changing derivation directions[2], changing the mode between moving and copying, quitting with saving the proof fragments data, and quitting without saving the proof fragments data.

Including these operations, the manipulation operations will be invoked from icons and menus displayed on editing and pop-up windows. In other words, the interface design of this system is GUI-based; i.e. menus, pop-up windows, mouse-oriented operations, and so on.

(2) EUODHILOS-II: Implementation on Emacs

One of the purposes of developing the second system is to provide a system on a popularly used platform so that a large number of people can use EUODHILOS systems. Considering the issues listed below, we took GNU Emacs[109] as the platform for implementing this system, which we call EUODHILOS-II, and implemented in Emacs Lisp.

- **Uniformity of Operations:** It is a great advantage for users to operate the new system with those operations which they use everyday and are familiar with. Various applications are developed on GNU Emacs, so that a lot of people use it every day. Thus it is easy to learn how to operate including the standard text editing features.

- **Portability:** In order to achieve the portability and easy circulation of the data, we need to choose a platform that runs on as many machines and operating systems as possible. GNU Emacs has been used on various environment and we can implement the extension functions by using only Emacs Lisp. Thus the system will be able to run as many machines as GNU Emacs runs.

- **Customization and Extensibility:** Various customizations should be done on an interactive system like G-RAS. On GNU Emacs, the user can easily customize various functions just by using Emacs Lisp. It is also fairly easy for the users to extend the manipulating functions by writing and adding them to the system.

[2]In the figure the arrow is downward, which means we are in the forward-derivation mode.

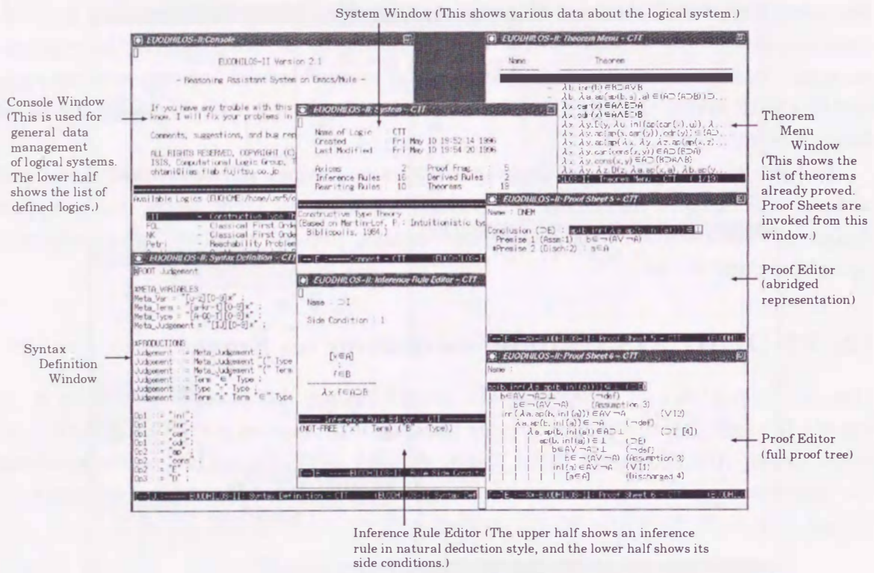


Figure 3.4: Example EUODHILOS-II Screen

- **Localization of Environment:** It is necessary for G-RAS systems to launch some number of applications at the same time so that it is possible to edit definitions of logical systems and editing proofs in parallel. It is easily implemented by using buffer-local variables of Emacs Lisp because they are local in the sense that a variable in one buffer is different from the variable with the same name in another buffer. Such concept is similar to the concept of capsulation in object-oriented programming languages.

An example EUODHILOS-II screen is shown in Figure 3.4. At the top left is the console window, which shows the list of logics already defined. New logics can be created by typing commands in this window. By selecting a logic from this list the system window corresponding to the selected logic is invoked and appears on the screen. The system window for the logic CTT(Constructive Type Theory) is in the center of the screen. All the operations concerning the logic are started by opening the corresponding window from the system window. All the windows for syntax

definition (the bottom left corner), inference rule editor (bottom center), theorems (top right), and proof editors (right middle and right bottom) are created from the system window.

Roughly speaking, the interface design of EUODHILOS-II is keyboard-oriented; basically all the operations can be performed through inputting command keys from the keyboard.

3.4 Chapter Summary

In this chapter we have developed a general system organization model called the EUODHILOS architecture, which is designed to fulfill the requirements to the G-RAS systems by considering the the conceptual human reasoning process model. We also verify its feasibility by implementing two systems on different platforms. These systems were designed based on the common architecture and by considering the features of the implementation languages and platforms. By taking this approach for designing and implementing G-RAS systems, we can discriminate the features that come from the G-RAS architectures and those from the implementation environment and the design policy.

To summarize the differences of EUODHILOS-I, which is implemented on PSI, and EUODHILOS-II, which is implemented on Emacs, including other aspects, we list up:

- **General Interface Design:** Considering the difference of the platform their basic interface designing policies are different. EUODHILOS-I is implemented as a menu-oriented system by considering the SIMPOS operating system on PSI machines. Many operations are derived by selecting an appropriate items in the menus. On the other hand, because EUODHILOS-II is implemented on top of Emacs/Epoch/Mule systems where lot of supporting functions has been prepared for keyboard operations, it is designed so that most of the operations can be performed by the commands given by the keyboard. Despite such differences, these two systems can be operated in a similar way because their conceptual operational models are common.

- **Language Description:** The language description in EUODHILOS-I is specified in the DCGo notation which is an extension to the definite clause grammar(DCG) notation, which is one of the most popularly-used notation for

Prolog-based systems. For parsing, the system uses a bottom-up parsing mechanism called BUP[57]. On the other hand, EUODHILOS-II uses the BNF(Backus-Naur Form) and uses the Earley's efficient top-down parsing algorithm[20]. Because of these differences the language description frameworks are different. However DCG notation is an extension to context-free grammar framework most syntax rule descriptions are somewhat similar from each other.

- **Parsing and Unparsing:** The difference of this feature also comes from the differences of the implementation languages and their processing environments. For Prolog-based systems it is fairly easy to generate a new program. Thus EUODHILOS-I takes the method of generating a parser and an unparser from the common description in DCGo notation. On the other hand, in EUODHILOS-II, it is more preferable to generate only the parse data and fixed parser and unparser use this data and perform their operations. This is the major reason why their parsing and unparsing mechanisms are different.

- **Proof Representation:** EUODHILOS-I deals with full tree-formed proofs on sheets of thought, whereas EUODHILOS-II deals with a couple of different display styles of proofs that are also in tree-form.

- **Tactics and Tacticals:** In this aspect the two implementations are different. EUODHILOS-I focuses on to give a sufficient supporting functions for user-initiated reasoning style. EUODHILOS-II, on the other hand, extends this framework so that it supports wider reasoning styles in the sense that it assists from the user-initiated step-by-step proof constructions to semi-automated proof constructions. Thus the differences basically comes from the differences of system designing policies of these two systems.

- **Interface Design:** Because of the differences of the basic policy for the interface of the systems, their primitive operations are different. EUODHILOS-I is based on GUIs such as the mouse, menus, pop-up windows, while EUODHILOS-II is based on key-inputting from the keyboard. This difference comes from the difference of platforms and from the differences of design policies.

Chapter 4

Language System Specification

The aim of this chapter is to establish a language system description framework for building up a wide variety of formal systems, which is the first component of a logical system description. We will deal with the other component, the derivation system, in Chapter 5.

In order to realize a user-friendly G-RAS system, it is quite important to design the well-equipped facility for providing the assisting environment on which the user is able to describe the objects and concepts of the target domain in his or her own way of representation and is also able to deal with these objects and concepts in the style which he or she is familiar with and thus easy to manipulate. A EUODHILOS system pursues its facilities based on this viewpoint. The role of the language system is to enable the user to specify the construction rules for terms, formulas, and other expressions used in the logic according to his or her own preference and decision. Therefore the feature for specifying the language system is one of the most important ones in our system.

A language system of EUODHILOS consists of the symbol definition and syntax specification. The facility for defining a new symbol provided by the system is as follows. In order to set up for a new symbol to be used in reasoning on EUODHILOS-I, the user (i) defines the symbol with the font editor, (ii) assigns the symbol on the software keyboard, and (ii) inputs the symbol by typing the key on which the symbol is assigned[66]. On EUODHILOS-II, the user (i) defines the symbol with the font editor, (ii) assigns the symbol on the key-sequence for it, and (iii) inputs the symbol by typing the key sequence on which the symbol is assigned[80]. See the description in Section 6.1.3 and the screen image in Figure 6.1 how new symbols are inputted. See also [63; 64; 66] for detail.

In the rest of this chapter, we mainly deal with the syntax description method with the following organization. In Section 4.1, we present a method of syntax specification written in the DCGo notation, which consists of syntax rules in DCG(Definite Clause Grammar) notation[87] and a constructor declaration. This framework is rich enough for specifying the syntax structures that beyond the context-free grammar by using arguments attached to nonterminals. It is also intuitive and thus easy to use because its description is a small extension to the context-free grammar. In Section 4.2, we present a procedure for generating a parser and an unparser from a specification in DCGo notation. A parser is a program that transforms the external string expression to its corresponding internal representation, whereas an unparser, often called a pretty printer, generates string expressions from the internal representations of logical formulas. By taking this mechanism for generating the both programs from a common description, the user is free from worrying about their mismatching problem, which will often occur if these two programs are developed separately. In Section 4.3, we present another method of syntax description which supposes that the external string expressions are parsed by a top-down parser. The parsing algorithm is a well-known efficient one for context-free grammars developed by Earley[20]. This is an example that implementation details are different by considering the features of the platform for realizing a function expected in the generic concept model of the system. In Section 4.4, we present an extension feature of specifying the structures of bind variable and its scope to the framework based on the context-free grammar notation. Such a feature will contribute to the system to be more adaptive to the user's preferred styles. In Section 4.5, we present some application examples and demonstrate how the parser and unparser framework can be effectively used. Lastly in Section 4.6, we summarize the discussions in this chapter.

4.1 Syntax Specification for Language System

The aim of this section is to present the DCGo notation that is easy-to-write and powerful enough for describing the syntax of a variety of formal systems. We will achieve this aim as follows. In Section 4.1.1, we define the concepts of parser and unparser programs and discuss what requirements should be put for syntax description framework that allows the automatic generations of parser and unparser at the same time. In Section 4.1.2, we describe the DCG(Definite Clause Grammar) notation, which is the notation on which the DCGo notation depends. In Section 4.1.3, we discuss the format of the internal representation of a logical expression, which is

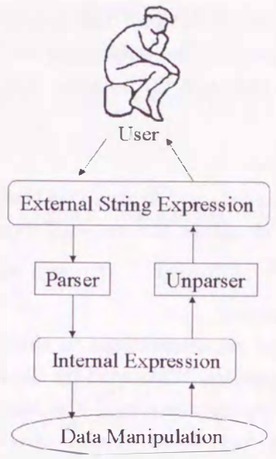


Figure 4.1: Relationship between Parser and Unparser

supposed to be a representation of the essential part of the logical expression. In Section 4.1.4, we put a couple of requirements on the syntax specifications for dealing with the expressions in an appropriate way. And, in Section 4.1.5, we give the DCGo notation that satisfies the requirements to the syntax specification framework for automatic generation of parser and unparser. The actual generation mechanism will be discussed and described in Section 4.2.

4.1.1 Parser and Unparser

Generally, the strings used in interfacing systems and humans have some kind of structure. For example, logical expressions and computer programs have structure of their own. Considering this fact, the internally represented expressions should be processed as structured data.

Consider also that the following two formulas "$\forall x.A(x)$" and "$\forall x.(A(x))$" are literally different from each other, while obviously they denote the same meaning. Therefore it is desirable that these two expressions are translated into exactly the same internal representation.

In order to deal with the external and internal representations, we need the parser, which translates from the external expressions to the internal ones, and the unparser,

which translates in the opposite direction. The functions of the parser and the unparser are illustrated in Figure 4.1. The parser and the unparser are important parts of the interface between the user and the system. They have to meet the following conditions:

(1) Conditions for the Parser:
It must accept only those external expressions that meet the language definition. Furthermore it should translate the external expressions into the appropriate internal representations.

(2) Conditions for the Unparser:
It translates the internal representations to their appropriate external expressions. The external expression translated by the unparser should be translated into the same internal representations if it is translated by the parser. On the other hand, the external expression which is obtained by the unparser from the internal representation that is obtained from the original external expression translated by the parser are not necessarily identical each other.

If a programmer writes the parser and the unparser manually, it would be a hard task because he or she has to take much attention to the programming so that the two programs satisfy the above conditions. Therefore it is preferable that the user describes the syntax structure of the language and the system automatically generates the parser and the unparser of it from the description so that the above conditions are satisfied.

The first candidate to such a syntax description framework is the DCG(Definite Clause Grammar[87]). It is preferable because it is based on the context free grammar and thus the user can easily describe his or her intended syntax. Its parsing algorithm is also well known[57]. Also it is easy to add some extra features in this framework.

However, in order to extend its framework, the user has to describe the additional part as if he or she is writing a program in a programming language. As the result we see that:

- Writing easiness decreases because the user has to have much knowledge about the programming and the internal structures of the expressions.
- The amount of the description will increase a lot.

Thus DCG does not meet the criterion for the syntax description framework we have set. In the following part of this section, we set up the DCGo notation, which


```
term1(t) --> term1(t), imply, term2(t);
term1(T) --> term2(T);
imply --> "=>";
term2(t) --> term2(t), and, term3(t);
term2(T) --> term3(T);
and --> "/\";
..........
term5((T1,T2)) --> lambda, variable(T1), ".", term5(T2);
..........
variable(T) --> var_sym, ":", type(T);
labmda --> "lambda";
type(e) --> "e";
type(t) --> "t";
type((T1,T2)) --> "(", type(T1), ",", type(T2), ")";
var_sym --> "x"|"y"|"p";
p_sym1 --> "f"|"g";
```

Figure 4.2: Syntax Rule of Intensional Logic(part)

is an extension to DCG notation and intends to overcome the above problems of DCG, and the method of generating the parser and the unparser. A main part of the DCGo notation is that of the DCG notation with some additional specification about constructors.

In the rest of this subsection, the DCGo notation is described. It enables to generate the parser and the unparser automatically at the same time. For preparation, the DCG notation is explained in Section 4.1.2. Then the format of the internal representation is given in Section 4.1.3. In Section 4.1.4 we investigate what considerations are needed in order to generate the parsers and unparsers automatically. In Section 4.1.5, we give how the DCGo notation is organized, with putting highlight on the constructor declaration format. We also summarize the DCGo syntax definition framework of EUODHILOS-I by presenting the framework in BNF notation.

4.1.2 DCG Notation

Since the DCG notation is based on the context free grammar, it inherits the feature that it is easy to write and read syntax rules in this notation. On the other hand, the attachment of arguments to the non-terminal symbols and combination of

procedure calls is added as an extension to the context free grammar. With procedure calls, we can implement the exchange of data among different syntax rules and procedures to them including transferring some context-sensitive information.

In the pure context-free grammar framework, we can define syntax of well-used formal syntax like predicate logic, where syntactic structure is context-free, sharing of data among syntax rules is not required. However it is impossible to specify the syntax of intensional logic in context-free grammars because there are some context-matching conditions in the syntax definition of the intensional logic. In the DCG framework, we can specify the syntax rule that transmits data to other rules. For example, in the syntax rule

```
term5((T1,T2)) --> lambda, variable(T1), ".", term5(T2);
```

in Figure 4.2, the argument `(T1,T2)` of nonterminal symbol `term5` carries the data `T1` and `T2`, and they specify the syntactic relationship between the argument of `term5` in the left part and the arguments of the nonterminals `variable` and `term5` in the right part.

4.1.3 The Format of the Internal Expression

The structural expressions consist of the composite expressions, which are constructed by combining subexpressions, and the elementary expressions, which can not be divided into smaller components. We take the standpoint that each composite expression has a constructor in its subexpressions. For example, we read the expression

$$\forall x.A(x)$$

as "the formula that the universal quantifier $\forall$ with parameter x is applied to the statement that 'A of x is satisfied'". This reading can be structurally expressed like:

```
["∀", "x",["A", "x"]].
```

The general form for expressing the internal representation is defined as follows.

$$[Constructor, Argument_1, \ldots, Argument_n].$$

The arguments of the internal representation are also internal representations, thus this concept is recursively defined.

The internal representation for an elementary external expression is the string expression of the external expression itself.

Structural information can be transferred among the syntax rules by using the arguments of the nonterminal symbols in the DCG notation. For example, the syntax rule that expresses the composition of the term having the type "t",

```
term2(t)  -->  term2(t), and, term3(t);
```

can be translated into the following DCG rule which has an argument that carries the structural information.

```
term2([AND, T2, T3], t)  -->  term2(T2, t), and(AND), term3(T3, t);
```

This syntax rule creates the internal representation in the left side by combining the structural information of the nonterminals "`term2`", "`and`", and "`term3`" in the right side.

4.1.4 Requirements on Syntax Description

At first, we would like to define some notations as preparation for the following descriptions. There are two types of symbols in syntax descriptions. The symbols in the first type are the essential components for the internal representation like the nonterminal and terminal symbols such as the constructor and its arguments. We call such symbols by "major symbols." The another type of symbols are those that are used not for essential components but are used auxiliary such as for ease of recognition. We call such symbols by "auxiliary symbols."

For example, in the next syntax rule:

```
term5((T1,T2)) --> lambda, variable(T1), ".", term5(T2);
```

"`lambda`" is the constructor and "`variable`" and "`term5`" are its arguments. Thus these three components are the major symbols and the component "`.`" is an auxiliary symbol.

In order to automatically generate the parser and the unparser we have to describe the following three types of information. The pure DCG notation is not good enough for describing the syntax for automatic generations of parser and unparser because it has no feature of specifying these information.

(C-1) Specifying the difference between the major symbols and the auxiliary symbols: Clarify which components are to be used for the internal representation and which are the auxiliary components.

(C-2) Specifying which is the constructor and which are its arguments: Denote the constructor among the major symbols. The rest of the major symbols will be used as arguments.

(C-3) Specifying the difference between the operators and predicative function symbols, and the difference of the priorities of operators.

The constructors that are used as pre-fix, in-fix, and post-fix operators are called the operators. An operator needs to have the priority because without it, ambiguity could occur when more than one operators are used and the parentheses are omitted.

For example, suppose the constructors "∧" and "⊃" are operators. The internal representation

```
["∧",A,B]
```

is translated into

```
"A∧B"
```

without ambiguity.

On the other hand, in the following example

```
["∧",["⊃",A,B],C]
```

where two operators appear, will be translated into

```
"A⊃B∧C".
```

This is ambiguous because it can also be interpreted like

```
["⊃",A,["∧",B,C]].
```

To avoid such an ambiguity the unparser should generate the expression with parentheses like

```
"(A⊃B)∧C".
```

We call the constructors the "predicative function symbols" if they are used in the expression for ordinary predicate and function symbols like "$p(x)$".

If we define the constructors "∧" and "⊃" as the predicative function symbols, the internal representation

```
["∧",["⊃",A,B],C]
```

will be translated into the string expression

```
"∧(⊃(A,B),C)"
```

so that no ambiguities occur even without operator priorities and parentheses for indicating the parsing orders.

4.1.5 DCGo Notation

The DCGo notation is the one obtained by adding the features of constructor declaration to the DCG notation.

The constructors are declared in the following format:

```
with_priority
```

$O_{11}, O_{12}, \ldots, O_{1n(1)};$

$\ldots$

$O_{m1}, O_{m2}, \ldots, O_{mn(m)};$

```
without_priority
```

$P_1, P_2, \ldots, P_l.$

where O_{ij} are operators and P_k are predicative function symbols.

In the constructor declaration, constructors are specified by discriminating the predicative function symbols and the operators. The priorities of O_{i*} are higher than O_{j*} if $i < j$, and the operators O_{ik} and O_{ih} for any h and k have the same priority.

Figure 4.3 illustrates a part of constructor definition of the intensional logic.

There are two types of syntax rules in the DCGo notation. The first type is the type only one terminal symbol appears in the right side. We call such a rule the "dictionary type". The other one is called the "rule type". In the DCGo notation the rule type rules should meet the condition that

> "If more than one major symbols appear in the right side of a rule, exactly one of them must be a constructor."

```
with_priority
   ":", ",";
   (not, lambda);
   and;
without_priority
   p_sym1.
```

Figure 4.3: Constructor Definition of Intensional Logic (part)

This condition is necessary because the internal representation has one constructor if it is constructed at all.

By categorizing the major symbols into nonterminal symbols and constructors, and defining auxiliary symbols as those nonterminals that are not constructors, DCGo notation satisfies the condition (C-1) in Section 4.1.4. It also satisfies the condition (C-2) because only the major symbols which are not constructors can be used as arguments. Furthermore, in the constructor definition, the operators and predicative function symbols are discriminated, and priorities of the constructors are specified. Thus the DCGo notation also satisfies the condition (C-3). As the conclusion, we say that the parser and the unparser can be automatically generated from the syntax description in the DCGo notation because the conditions from (C-1) to (C-3) are satisfied.

The DCGo notation has the following requirements besides the above requirements. These are the weak limitations, i.e. the limitations only for efficiency of processing, and not the crucial ones for descriptions of the syntax.

- The rules should be cycle-free:
  A cycle-free rules are the ones which have no rules like
  $S \rightarrow S$
  nor the cyclic rules like
  $S \rightarrow S_1$
  $S_1 \rightarrow S_2$
  ...
  $S_{n-1} \rightarrow S_n$.
- The rules should not include ε-rules:
  An ε-rule is a rule which has empty right side; i.e. no terminals and nonterminals

appear in the right side of the rule. Such a rule is equivalent to the dictionary type rule that has sole null string in the right side. This requirement comes from that we use a bottom-up parser for parsing the string expressions, where the parser should generate all the possible parsing data in every place between two characters; which is too much in terms of efficiency. Thus even this requirement is not an crucial one, it is better to avoid using such rules for efficient parsing.

Description Formats

The DCGo notation described in this section that consists of the rules in DCG notation and the constructor definition formally specified as follows in the BNF[36] notation.

(i) Notations in the BNF Notation

- "::=" indicates that the symbol in the left side is defined by the right side. The symbol "|" in the right side specifies that the expression in either side can be used for the definition.
- "X" indicates a terminal symbol X.
- "{X}" indicates the repetition of X including zero times; i.e. no X.
- "[X]" indicates one or no X; i.e. optional occurrence of X.

(ii) Syntax Rule and Constructor Definition Format in BNF Notation

(a) Syntax Definition:

```
<Syntax and Constructor Definition> ::=
     <Syntax Definition> <Constructor Definition> "."
```

(b) Syntax Rule Definition:

```
<Syntax Definition> ::=
    <Syntax Rule> {";"<Syntax Rule>}
<Syntax Rule> ::=
    <Non-Terminal Symbol> "-->" <Right Side>
<Right Side> ::=
```

```
    <Clause> {"," <Syntax Element>}
    | <Right Side> {"|" <Right Side>}
<Clause> ::=
    <Non-Terminal Symbol>
    | <Terminal Symbol>
    | <Operator>
    | <Predicative Function Symbol>
<Syntax Element> ::=
    <Clause>
    | <Call Clause>
<Non-Terminal Symbol> ::=
    <Atom>
    | <Complex Term>
<Terminal Symbol> ::= <String>
<Call Clause> ::=
    "call(" <ESP Method List> ")"
<ESP Method List> ::=
    <ESP Method> {"," <ESP Method>}
<ESP Method> ::=
    <Complex Term>
    | <Operator-Application Term>
```

(c) Constructor Definition:

```
<Constructor Definition> ::=
    "with_priority" <Operator Definition>
    "without_priority" <Predicative Function Symbol Definition> "."
<Operator Definition> ::=
    <Operators in the Same Priority>
      {";" <Operators in the Same Priority>} ";"
<Operators in the Same Priority> ::=
    "(" <Operator> {"," <Operator>} ")"
<Operator> ::=
    <Non-Terminal Symbol>
    | <Terminal Symbol>
<Predicative Function Symbol Definition> ::=
```

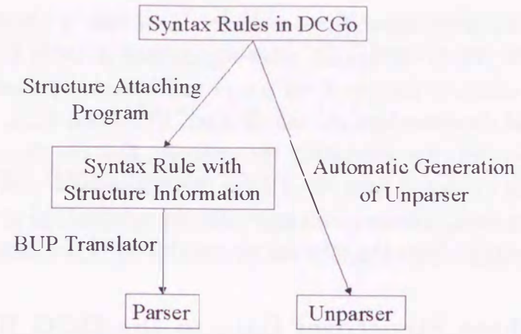



Figure 4.4: Parser and Unparser Generation from Syntax Rules

```
    <Predicative Function Symbol>
      {"," <Predicative Function Symbol>}
<Predicative Function Symbol> ::=
    <Non-Terminal Symbol>
    | <Terminal Symbol> See the ESP Manual[12] for the definitions of <Atom>,

<Complex Term>, <String>, and <Operator-Application Term>.
```

4.2 Automatic Generation of Parser and Unparser

In this section, we present how parsers and unparsers are generated from the common description written in the DCGo notation. Figure 4.4 illustrates how the parser and the unparser are automatically generated from the syntax description written in DCGo notation. For generating the parser, firstly, the syntax rules with the structure information is created from the syntax rules in DCGo notation by the structure attaching program, then the parser is generated from this data by the BUP[57] translator. For the unparser, an unparser generator generates the unparser directly from the original DCGo syntax rules.

The rest of this section is organized as follows. In Section 4.2.1, we present how the structural data are attached to the original DCG rules in reference to the constructor declaration. We describe the procedure of attaching the structural data to the given DCG rules and creates the new DCG rules. The roles of the parser are not only to check the given character string if it is syntactically correct but also to generate

the internal representation data if it is correct. In Section 4.2.2, we deal with the parsing algorithms. From the syntax rules represented in DCG notation, a parser is generated. We compare the top-down parser and bottom-up parser by discussing the advantages and disadvantages of each of them. In Section 4.2.3, we describe how unparsers are generated. For generating the unparser, first the characterization data for the unparser is extracted from the DCGo description and then the unparser is generated from the data. The unparser generates the external string expressions that are syntactically correct from the internal representation of the logical expressions.

4.2.1 Attaching Structural Data to the DCG Rules

Here we show how the structural representations and the syntax rules correspond to each other. We divide the structural representations into two categories: elementary representations and compound representations. The elementary representations are translated into dictionary type syntax rules. The compound representations are translated into rule type syntax rules that have more than one major symbols in the right side of the rules.

A rule type syntax rule of the form

```
term1 --> term2;
```

having single nonterminal symbol in the right side indicates that the expressions of the type `term2` can be interpreted as of the type `term1`.

The structural data are attached as the arguments of nonterminal symbols, where the internal representations are expressed as metavariables. The metavariables in the right side are instantiated in the Prolog unification mechanism and are used as the constructor and the arguments of the resulting internal representation corresponding to the rule.

The translation is performed as follows:

(i) For dictionary-type syntax rules:

This type corresponds to the elementary expression. The elementary expression in the right side is used also as the internal representation. For example, in the following rule sample of

```
type(e) --> "e";
```


the terminal symbol `"e"` itself is attached to the first argument of the nonterminal in the left side as the internal representation of `"e"`. Thus the resulting syntax rule is:

```
type("e",e) --> "e";.
```

(ii) For rule-type syntax rules:

The syntax rule of this type is classified further by the number of major symbols in the right side; i.e. the number of components necessary for constructing the internal representation.

(a) If the major symbol is only one:
The internal representation is used also as the resulting internal representation. For example,

```
term2 --> term3;
```

is translated to

```
term2(A) --> term3(A);
```

where the internal representation for `term3` is given to its argument `A` and it is used as the argument of the resulting `term2`.

(b) If the major symbols are more than one:
From the condition we impose at Section 4.1.5, one of the major symbols should be declared as constructor. From the constructor declaration we can say which major symbol is the constructor. We take the internal representation of the constructor as the constructor of the resulting internal representation and those of the rest major symbols are used as the arguments of the representation. For example, let us suppose, following to the definition in Figure 4.3, the symbol `":"` is the constructor of this construction.

```
variable(T) --> var_sym, ":", type(T);
```

Let us use "`VAR`" and "`TYPE`" as the metavariables for the nonterminals `var_sym` and `type`, respectively. Then the above rule is translated into the following DCG rule:

```
variable([":", VAR, TYPE], T)
        --> var_sym(VAR), ":", type(TYPE, T);
```

Next consider the syntax rule

```
term5((T1, T2)) --> lambda, variable(T1), ".", term5(T2);
```

including an auxiliary symbol ".". The internal representation does not include any part corresponding this symbol. Also from the definition in Figure 4.3, lambda is a constructor. Let us use the metavariables "LAMBDA", "VAR", and "Term5" for the internal representations for lambda, variable, and term5, respectively. Then we have the following syntax rule.

```
term5([LAMBDA, VAR, Term5], (T1,T2))
--> lambda(LAMBDA), variable(VAR, T1), ".", term5(Term5, T2);
```

See that the translation rule is the same as the example above except the corresponding argument for the auxiliary component does not appear in the resulting internal representation.

Following to the translation rule described in this section, the syntax rules described in the DCGo notation are translated into the syntax rules with the structure information in the DCG notation.

4.2.2 Automatic Generation: Parser

By the translator described in the previous section, the syntax rules written in the DCGo notation are translated into the syntax rules with the structural data in the DCG notation.

The top-down parser for DCG[13; 87] and the bottom-up parser BUP[57] are well known as the Prolog-based parsers. With the top-down parser, the infinite loop can occur in parsing the left-recursive syntax rules[13; 57]. We take the bottom-up parser BUP as the basic algorithm because left-recursive syntax rules should not be prohibited as they will appear frequently when the users write their syntax rules. Also the effective algorithms that translate the syntax rules written in the DCG notation into BUP parser are known.

The syntax rules that are attached with structural data are given to the BUP parser translator and the BUP parser program in Prolog is created. In Figure 4.5, part of the resulting BUP parser program translated from the DCGo description of the intensional logic is illustrated.


```
parse(G,A,S) :- goal(G,A,S,[]).
goal(G,A,S1,Sn):-
  (dict(N,X,S1,Sn,C),link(N,G),call(C);
  e_rule(N,G,X),S1=S2),
  P=..[N,G,X,A,S2,Sn],call(P).

term1(term1,A,A,S,S).
term1(G,[T1,t],X,S1,Sn):-
  link(term2,G),
  goal(or,[OR],S1,S2),
  goal(term2,[T2,t],S2,S3),
  term2(G,[[OR,T1,T2]],X,S3,Sn).
term2(term2,A,A,S,S).
term2(G,[T2,t],X,S1,Sn):-
  link(term2,G),
  goal(and,[AND],S1,S2),
  goal(term3,[T3,t],S2,S3),
  term2(G,[[AND,T2,T3]],X,S3,Sn).
term3(term3,A,A,S,S).
term3(G,[A,T],X,S1,Sn):-
  link(term2,G),
  term2(G,[A,T],X,S1,Sn).
  ..................

dict(and,["^"],["^",S1],Sn,true).

  ..................
```

Figure 4.5: BUP Parser on Intensional Logic(part)

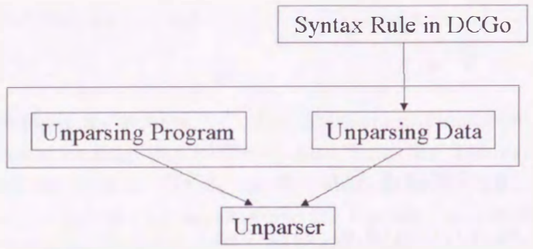


Figure 4.6: Unparser Generation for Syntax Rules in DCGo Notation

4.2.3 Automatic Generation: Unparser

The internal representation is the form used for manipulating in the system. Since the auxiliary symbols are not essential internally, they do not appear in the internal representation. However, once the internal data are translated into the external use, they should be translated so that the auxiliary parts are added. Figure 4.6 illustrates how the unparser is generated.

The core part of the unparser are common among all the unparsers. Only the data for unparsing depend on the syntax rules described in the DCGo notation.

(1) Components of the unparsing data:

The unparsing data are created from the rule type rules that have the constructors. If the constructor of the rule is declared as the operator with priority, i.e. the operator declared in the `with_priority` part, its unparsing data consists of three components:

- String-Generation List
- Position of the Constructor
- Priority

If the constructor of the rule is the predicative function symbol, or the operator declared in the `without_priority`, its unparsing data consists of the following two components:

- String-Generation List
- Position of the Constructor

A string-generation list represents which components are arranged in the string expression and consists of the elements corresponding to the components of the right side of the rule. If the corresponding component is a nonterminal symbol then the element is a variable for representing the string expression for the nonterminal symbol, and if the corresponding component is a terminal symbol then the element is the terminal symbol itself. The position of the constructor specifies which element corresponds to the constructor, which appears as the predicate name in the internal representation. The priority data can be obtained from the constructor declaration and will be used for deciding if the parentheses are necessary or not.

We will illustrate the procedure of generating the unparsing data from the syntax rules by using the following example syntax rule.

```
term5((T1,T2)) --> lambda, variable(T1), ".", term5(T2);
```

The right side of this rule has four components; the first, second and fourth elements are nonterminal symbols, and the third one is a terminal symbol, or a string. The unparsing data for this rule is as follows.

- String Generation List: `[LAMBDA, VAR, ".", T5]`
- Constructor Position: 1
- Priority: 2

In the string generation list, the corresponding element for the nonterminal symbols are variables; namely `LAMBDA`, `VAR`, and `T5`. The element corresponding to the terminal, i.e. `".",` is the string `".".` From the constructor definition in Figure 4.3, the nonterminal symbol `lambda` is a constructor with priority 2. Thus the priority is set to 2. This nonterminal `lambda` appears as the first component of the right side of the rule. Thus the constructor position is 1.

(2) Unparsing Procedure

We will describe the unparsing procedure by separating the cases of (i) if the internal representation is an elementary expression, and (ii) if it is a compound expression. For the compound expressions the procedure uses the unparsing data described in the previous section.

(i) For Elementary Expressions:

The internal representations for elementary expressions are the expressions themselves. Thus the internal representation itself is to be used as the unparsed string.

(ii) For Compound Expressions:

Since the compound expressions are defined recursively, the unparsing procedure is also applied recursively and we suppose all the arguments have the corresponding unparsed strings, or external expressions.

If the internal representation given as the argument to this procedure is a compound expression, it may be necessary to put parentheses in order to avoid ambiguities of the generated character string. It is not necessary to put parentheses if the constructor is a predicative function symbol. On the other hand, if it is an operator, the procedure has to compare the priorities of the operator of the argument (i.e. child operator) and the operator in the original operator (i.e. parent operator). The need of putting parentheses is different in the cases when the priority of the parent operator is greater than that of the child operator and when the priorities of them are in the opposite order.

(a) If the priority of the parent operator $<$ the priority of the child operator:

In this case the procedure does not put parentheses to the arguments. For example, suppose the arguments have been unparsed like

```
["λ", "x:t", "x:t"].
```

The arguments are the results of unparsing the internal representations having `":"` as operators. Thus the operator `"."` is the child operator, the priority of which is 1 from the constructor definition. The parent operator is `lambda`, which has priority 2. The operator having smaller priority number is stronger in terms of binding power,

$$\text{parent's priority} < \text{child's priority}$$

holds, thus it is not necessary to put parentheses.

(b) If the priority of the child operator $\leq$ the priority of the parent operator:
In this case the procedure puts parentheses around the argument. For example,

in the following

```
["λ", "x:t", "x:t∧y:t"]
```

the priority of the parent operator λ is 2 and that of the child operator $\wedge$ is 3, thus the priority of the child operator $\leq$ the priority of the parent operator. In this way we get the unparsed character string

```
"λx:t.(x:t∧y:t)".
```

The procedures of the two cases of translation are illustrated in Figure 4.7 and Figure 4.8.

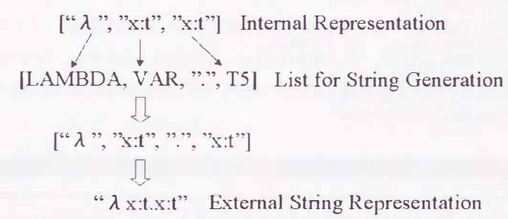


Figure 4.7: Example of Unparsing `["λ","x:t","x:t"]`

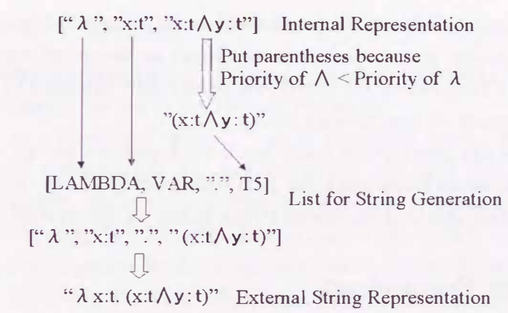


Figure 4.8: Example of Unparsing `["λ","x:t","x:t∧y:t"]`

```
EUODHILOS-II: Syntax Definition - Pred
%ROOT Formula

%META_VARIABLES
Identifier = "[A-Z][A-Z0-9]*" ;
Variable = "[s-z][0-9]*" ;
Constant = "[a-e][0-9]*" ;
FunctIdentifier = "[f-h][0-9]*" ;

%PRODUCTIONS
Term ::= Variable ;
Term ::= Constant ;
Term ::= Term "/" Term ;
Term ::= FunctIdentifier "(" ListOf_Term ")" ;
ListOf_Term ::= Term ;
ListOf_Term ::= Term "," ListOf_Term ;
AtomicFormula ::= "⊥" ;
AtomicFormula ::= Identifier ;
AtomicFormula ::= Identifier "(" ListOf_Term ")" ;
AtomicFormula ::= Term "=" Term ;
Formula ::= AtomicFormula ;
Formula ::= "(" Formula ")" ;
Formula ::= "¬" Formula ;
Formula ::= "∀" @Variable "." [ Formula ] ;
Formula ::= "∃" @Variable "." [ Formula ] ;
Formula ::= Formula "∧" Formula ;
Formula ::= Formula "∨" Formula ;
Formula ::= Formula "⊃" Formula ;

[--]J.:-----EUODHILOS-II: Syntax Definition - Pred      (EUODHILOS-II Syntax Def
```

Figure 4.9: Syntax Definition Window

4.3 Syntax Specification for Top-Down-Parsing

The syntax specification framework presented so far in this chapter intends to be parsed by the bottom-up parser that was described in the previous section. In this section, we present a framework for top-down parser and discuss the differences that come from the difference of the parsing styles.

We first describe the description framework for top-down parser which is applied to the syntax specification framework for EUODHILOS-II. Then we discuss the processing mechanism and internal representations taken in the system.

(1) Description Framework

This is the framework of the context-free grammar in the BNF-notation for the syntax description, which is easy to write and is expressive enough for a wide variety of logics. Figure 4.9 shows an example of the syntax definition window.

The syntax definition consists of three parts: *ROOT*, *META_VARIABLES*, and *PRODUCTIONS*. The *ROOT* part tells the top-down parser[20] which symbol is the starting symbol for expression parsing, which specifies the top-most string expression used in the reasoning step. In most logical systems this is called a "formula." In some systems, the basic expressions used in reasoning have different nonterminal symbols. For example in Martin-Löf's type theory[55], it is called a "judgment."

The strings defined in the *META_VARIABLES* part are parsed as metavariables. They are given in regular expressions. Metavariables are place holders that represent expressions having the same syntax category and can be instantiated at any time in reasoning. These are used to make schematic expressions to assist in axiom description, rule description, and proving. Using metavariables, users can start proving without specifying the details of the formulas. The metavariables are cumulatively instantiated as the proving advances, and the final expression is obtained when the proof is completed. Metavariables are necessary in combining proofs and in using theorems and derived rules.

The string representations of the metavariables are given in regular expressions. For example:

```
Identifier = "[A-Z][A-Z0-9]*" ;
```

declares that "`Identifier`" is a metavariable, describing all strings beginning with a capital letter followed by an arbitrary number of capital letters or numerals.

The *PRODUCTIONS* part consists of one or more production rules. This is written in the BNF notation[36]. For example, the production rule:

```
Formula ::= Formula "∧" Formula ;
```

states that a sequence consisting of a string parsed as "`Formula`", the character "∧" and another "`Formula`" string is parsed as "`Formula`".

Substitution is represented with "/" as is normally used. This expression represents the partial substitution for free variables and will not rename the bound variables. For example, "`A(t/x)`" represents the logical formula that will be obtained from "`A`" by substituting "`t`" for some free occurrences of "`x`". This substitution expression will be used together with the metavariables in the definitions of axioms and inference rules. By using the side conditions `FREE-FOR` (Section 5.2) that specifies the substitutability, appropriate substitution will be automatically performed when it is appropriate in the process of proving (See the 6th step of deduction process of Section 6.2.4).

Furthermore, the production rule of the form

```
Formula ::= "(" Formula ")" ;
```

is considered to be the parenthesizing rule[1] so that redundant parentheses are removed in parsing and appropriate parentheses will be added according to the priorities of logical constructors when a logical internal expression is translated into external string expressions; i.e. when unparsed. In the syntax definitions the rule that appear earlier has stronger priority than those that follow for each syntax element.

The syntax analysis data will be generated from the syntax definition data when it is saved in a file. The generic parser refers the syntax analysis data and perform the syntax analysis. The parser use the Earley's algorithm[20] for analyzing context free grammar.

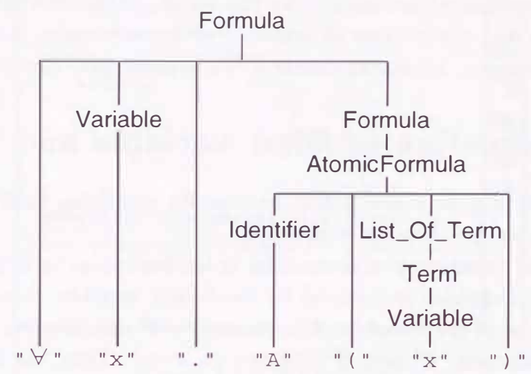
For definitions of languages it is a crucial problem to keep the consistency of the syntax definition and other data of the logical system, because a string once parsed correctly may be parsed in a different way with the user's intention because of some modifications of definitions of some data. In EUODHILOS-II, a function checks all the derivation systems and proof data by re-analysing the syntax of all the strings so that the consistency of the syntax definition and other data will be kept. If some serious errors are found, users can adjust the difference by editing the definition of syntax instantly.

(2) Internal Representation

As is shown in Figure 4.10, a formula is represented in a S-expression[115] that is directly coded from the syntax tree. The first element of the list denotes the production rule which is applied as the first rule to the given string. The rest elements are internal representations of child-nodes. For parenthesized expressions, redundant parentheses are removed. Further, the information relating to the bound variable and its scope with "@", "[" and "]" as was explained in Section 4.3 will be referred to when the inference rules are applied or when the side conditions are checked. It is not included in the internal representation of a formula. The processing method about bound variable and its scope is described in Section 5.2.2.

For the case of Figure 4.10, the first element of the list is the S-expression of the production rule of the form:

[1] "(" and ")" can be replaced with other parenthesizing pairs that are popularly used.




```
((Formula "∀" Variable "." Formula) ; Production Rule
  (terminal . "∀")          ; Internal Representation of "∀"
  (Variable . "x")          ; Internal Representation of "x"
  (terminal . ".")          ; Internal Representation of "."
  ((Formula AtomicFormula) ; The followings are for "A(x)"
    ((AtomicFormula
            Identifier "(" List_Of_Term ")")
     (Identifier . "A")
     (terminal . "(")
     ((List_Of_Term Term)
       ((Term Variable)
         (Variable . "x")))
     (terminal . ")"))))
```

Figure 4.10: Derivation Tree of "$\forall x.A(x)$" and its Internal Representation

```
Formula ::= "∀ " @Variable "." [ Formula ];
```

The rest part consists of "∀", followed by the internal representations "x" of a variable, ".", and followed by the internal representation of "A(x)" for "Formula".

By taking such an internal representations, the production rule that generates the data can be obtained instantly. As the result, substitutability checking and the substitutions and connections of system tree become easier in the substitution operation. Furthermore, no syntax checking is necessary with this representation.

4.4 Specification of Bind Variable and its Scope

In this section we propose a description method for specifying bind variable and its scope for the syntax specification framework.

The concept of quantification is essential in the first-order logic and other logical systems, where a quantifier is followed by the bound variable, then by the formula that is in the scope of the variable. The character "@" specifies the bound variable, and the square brackets "[" and "]" indicate its scope. Thus, the ordinary form of expression of quantifier "∀", in the example screen at Figure 4.9, would be as follows:

```
Formula ::= "∀" @Variable "." [ Formula ];
```

It declares that the string sequence that consists of the string that a symbol "∀" followed by a string represented by `Variable` followed by "." and `Formula` also represents a `Formula` expression. Here the string for `Variable` is a bound variable and that of `Formula` is its scope. With using this description we can define the scoping rule of $(\forall x \in A)B(x)$, which is difficult for ordinary logical framework by:

```
Formula ::= "(" "∀ " @Variable "∈ " Formula ")" [ Formula ];
```

It is also easy to define other binding structures different from the one extracted from logical systems. For example, the following defines the `let`-statement structure commonly used in functional programming languages[88]:

```
"let " @Variable "=" Expr " in " [ Stmt ];
```

4.5 Application Examples of Parser and Unparser

In this section we take up some examples and demonstrate how parser and unparser can be effectively used for realizing their functions.

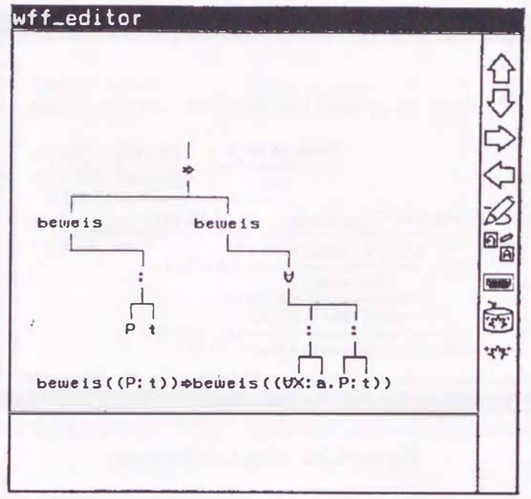


Figure 4.11: WFF-Editor

(1) Application to Well-Formed-Formula Editor

Figure 4.11 is a window of WFF-editor(Well-Formed-Formula Editor)[61; 66; 95; 112]. WFF-editor is a structure editor where users can edit complicated formulas(or logical expressions in general) with ease. The expression in the bottom line is the string(or external) expression of a formula. The parser is invoked as the user inputs this expression. The resulting parsed tree is displayed as is shown in the figure. Users can edit the formula by using the mouse and the icons. The unparser will be invoked when a component of the tree has changed and the new structure will be re-displayed.

(2) Application to Sheet-of-Thought

Figure 4.12 is a window of sheet-of-thought, where the users edit various number of proof fragments and try to construct the intended proofs. Users will, for example, input a formula when he or she wants to put an assumption or a goal. The parser is automatically invoked when a string expression is typed. The sheet-of-thought checks if the expression is an allowable one by invoking the parser. The unparser is called when a new application or connection of proof fragments are performed and

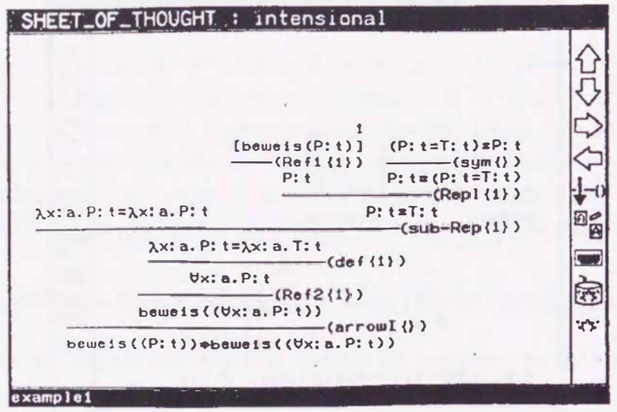


Figure 4.12: Sheet of Thought

one or more new internal expressions are created so that one or more new and old string expressions are generated. The sheet-of-thought forms the new tree expressions according to the changes of the internal representations and re-display the proof trees by using the string expressions generated from the unparser.

(3) Application to Syntax Checker

Due to the facility of automatic generation of parser and unparser, the burden of the users decreases a lot than that when users have to write them manually. However, as we used such facilities, we found that it is still not an easy task to define the intended syntax rules correctly. Two major causes will be notable on such difficulties:

(i) There are some errors in the syntax definition so that some external string expressions fail to be parsed correctly.

(ii) Due to the wrong specification of constructors because of lack of knowledge or misunderstanding, the structure described in the syntax definition is different from the intended structure of the user.

The syntax checker is developed to reduce the errors of the first type. Figure 4.13 is an example session for syntax checking. On this window the checker put the inputted string to the parser and see what syntax type it will match. Note that in EUO-DHILOS-I the parsing algorithm is the bottom-up type so that it is not necessary to

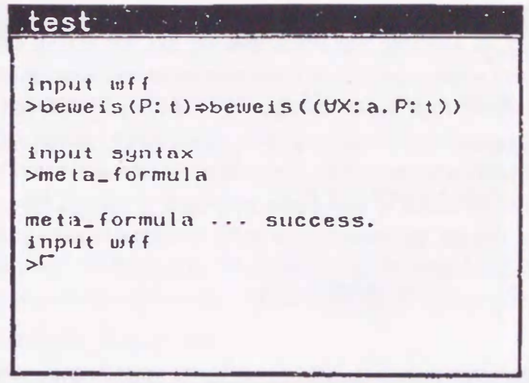


Figure 4.13: Syntax Checking Window for DCGo Notation

specify what syntax type the user intends. The system returns all the possible syntax type of the string or the parsing procedure fails. If the user specifies what syntax type he or she intends, just put the intended syntax after the input command as a parameter. The syntax checker returns if it is right or wrong.

WFF-editor is a relevant tool for checking and correcting the second type cause of the errors in syntax definition.

(4) Application to Syntax Checking for Top-Down-Parser

A syntax checking window for top-down parser is shown in Figure 4.14. Two types of checking are available in this syntax checking window. One type of them checks whether the given expression fits the given syntax category, and the other makes this check and also shows the parsing structure when the parsing ends successfully. In Figure 4.14, the parsed trees of the formula "$\forall x.A(x)$" is displayed.

The biggest difference of the syntax checking window of bottom-up parser and that of top-down parser is that in the latter one, the user has to specify what syntax category he or she is expecting for the given string expression, whereas in the former one, as was described, the parser is able to find the matching categories for the given string expression so that the user does not need to specify the expected syntax category to the parser.

```
EUODHILOS-II:Pred-Show Structure
Formula -->
  ∀
  *Variable* --> x
  .
  Formula -->
    AtomicFormula -->
      *Identifier* --> A
      (
      ListOf_Term -->
        Term -->
          *Variable* --> x
      )

[--]E :--%%-Mule:  EUODHILOS-II: Show Structure - Pred
```

Figure 4.14: Syntax Checking Window for Top-Down Parser

4.6 Chapter Summary

In this chapter, we have investigated a framework for language system description that is expressive enough and easy-to-write and easy-to-read, for EUODHILOS system. We have presented a framework for syntax description in the DCGo notation for genrating the bottom-up parser and unparser automatically. The DCGo notation consists of the rule specification part in the DCG notation and the constructor declaration. Since the parser and the unparser are generated from a common syntax description they are guaranteed to satisfy the conditions that are posed for consistent parsing and unparsing. We have presented another syntax description framework which is for a top-down parsing algorithm. We have presented a new description method for specifying the structures for bind variable and its scope. Some application examples were presented, where the parser and unparser are well-used for demonstrating the usefulness of the frameworks proposed in this chapter.

As parser generators, Yacc[41; 51] and Synthesizer Generator[91] are well known. However Yacc does not care about the mechanism for unparsing, probably because its main concern is the syntax analysis for compilers and other such types of systems. In the Synthesizer Generator the user has to describe both for parsing and unparsing, thus the consistency is up to the user, thus it does not meet our requirements.

The description frameworks are realizations of one conceptual architecture called

EUODHILOS. By reflecting the differences of platforms and designing policies the implemented mechanisms for syntax definition are different in these two systems. For evaluating the facilities of these systems the following aspects are important: (i) Representation: how the objects are represented, what is the amount of description, and so on, (ii) Operation: what process of operations is necessary for achieving the user's goal, what is the amount of operations, and so on, (iii) Flexibility: how is the system flexible enoufh so that it meets the user's way of thinking, doing things, and so on, (iv) Useful Supporting Facilities: what sort of supports can the users get from the system. We conclude this chapter for discussing the differences of these two systems on the following items with considering these evaluation viewpoints.

- Syntax Description Framework:
  On one system, syntax is specified in DCGo notation, which is an extension to DCG notation with constructor declaration. On the other system, syntax is specified in context-free grammar in BNF notation with additional specifications for root nonterminal symbol and metavariables definitions. Both frameworks are easy to describe and recognize for users. The amount of descriptions are also reasonable; especially considering one description is enough for both parsing and unparsing.

- Parsing and Unparsing:
  On one system, the system generates the parser and unparser program from syntax definition data for each logical system. On the other system, the data for syntax analysis are generated from the syntax definition and the generic parser that does not depend on specific logics performs the syntax analysis by referring the data. This difference affects to the efficiency for preparation for the parsing and unparsing, and the efficiency of the processes themselves. The method taken in the former one would need more time in generation process than that of the latter one. For the parsing and unparsing processing time, the method of the former is more efficient than that of the latter.

- Metavariable:
  Metavariables are place holders in various expressions that represent expressions having the same syntax category. These are used to make schematic expressions to assist in axiom description, rule description, and proving. This is one of the key features in proving. Using metavariables, users can start proving without specifying the details of the formulas. The metavariables are cumulatively instantiated as the proving proceeds, and the final expression is obtained when

the proof is completed. Superficially the metavariable specifications of the two systems are different. In the former system, nonterminals the names of which begin with "meta_" are considered to be metavariables. In the latter system, metavariables are defined in special parts in the syntax definition. However they are essentially the same. A metavariable represented in one system can be specified in the other system.

- Useful Supporting Facilities:
  For supporting the language system description both systems provide the syntax checking facility. We take up this issue here, because the checking facility for expressions is quite useful. From our experiments we recognize that users make a lot of mistakes in defining syntax which is beyond our expectation. This facility checks whether the given expression drops in the given syntax category, and optionally it also shows the parsed tree when it ends successfully. By using such a tool, users can correct errors in an early stage.

Chapter 5

Derivation System Specification

The aim of this chapter is to present a framework for specifying the structure of the target domain of the intended formal problem solving. Following to the general EUODHILOS architecture, a derivation system consists of axioms, inference rules, and rewriting rules in a natural-deduction style formalism. Because this framework is quite natural to users, it is easy to write, recognize, and correct errors of the specification.

The derivation system specifies the deductive structure of the properties that are satisfied in the target domain that the logic will represent. There are two types of derivation systems. In one type, all the properties are deduced from axioms, which are the initial postulates that are considered to be satisfied in the domain being modeled. The rules derive new properties from old ones, which are either axioms or theorems that are derived in the previous steps. The Hilbert style and the sequent style formulations use this type.

In the other type, the reasoning may start from assumptions, which are formulas put as working postulates at the beginning of the reasoning. Rules work also in this type. The results obtained in this type of deduction depend on the assumptions used in the deduction process. Assumptions must be discharged in order to get theorems, which do not depend on any assumptions. Natural deduction uses this type of deduction.

By comparison, the former one is more suitable for analyzing a logical system, like analyzing the cut-elimination property is to be dealt with in the sequent style formulation. The latter one is more suitable to human reasoning; that is, in the latter type, it is easier to start reasoning by putting postulates that might be used in the reasoning. It is also easier to find the rules that are applicable in a given situation.

The framework for specifying derivation systems of EUODHILOS architecture is taken so that these two types of deductions. We call it the natural deduction style formalism, where users may put assumptions to the inference rules. If the user wants to take the first type of system, he or she may formulate the rules so that they have no assumptions. If he or she wants to take the second type of system for formulation, he or she may use assumptions for the rules. See Chapter 7 for actual definition examples.

Based on the natural deduction style, the derivation systems can be formulated by combining axioms, inference rules, and rewriting rules in EUODHILOS so that the user can formulate a formal system by his or her own preference and by considering the characters of the formal system. Thus it is completely up to the user how to formulate a target domain including which type of deduction to choose and which style of formulation to take. It is worth to note here that in this framework, specifications are intuitively represented so that, like that of language system, it is easy to write and easy to recognize.

The rest of this chapter is organized as follows. In Section 5.1, we describe the specification framework for axioms, inference rules, and rewriting rules. In Section 5.2, we describe the specification method for side conditions and discuss some related topics. In Section 5.3, we conclude this chapter.

5.1 Specification of Derivation Systems

In this section we present the framework for specifying a derivation system and discuss how this framework meets the general EUODHILOS philosophy for assisting for usable G-RAS system facilities.

(1) Axiom

An axiom of EUODHILOS consists of a name, body expression, and optional side conditions. The side conditions, if given, are conditions which must be satisfied when the axiom is used in a proving process.

(2) Inference Rule

An inference rule consists of a name, conclusion, one or more premises, optional assumptions for each premise, and side conditions that are also optional to the rule.

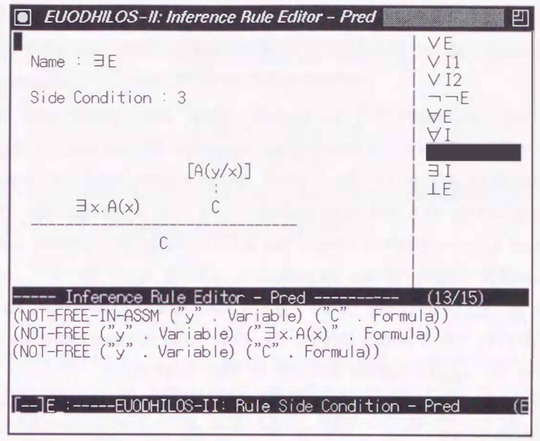


Figure 5.1: Inference Rule Window.

An inference rule is given in the general form:

$$\frac{\begin{array}{c}[A_{11}][A_{12}]\ldots \\ \vdots \\ Premise_1\end{array} \quad \cdots \quad \begin{array}{c}[A_{n1}][A_{n2}]\ldots \\ \vdots \\ Premise_n\end{array}}{Conclusion}\ Conditions$$

which states that the conclusion follows from the premises, which are derived under the assumptions that are located above the premises. The assumptions are discarded, or "discharged", when the rule is applied.

In Figure 5.1 we show an inference rule of first-order logic. The rule has the name "$\exists E$", conclusion "C", and two premises "$\exists x.A(x)$" and "C". The second premise "C" has the assumption "$A(y/x)$", which denotes the expression obtained from expression "A" with the substitution of "y" for the occurrences of "x". The rule also has three side conditions which are displayed in the lower pane. We will be back to this issue later in Section 5.2.

(3) Rewriting Rule

Rewriting rules are useful for handling equational reasoning often appearing in ordinary mathematical practice. A rewriting rule is specified with a pair of forms before and after rewriting in the following schematic format:

$$\frac{Upper\_expression}{Lower\_expression}.$$

A rewriting rule is considered to specify that the upper and lower expressions can be used interchangeably. The rule is applied to an expression when it has a subexpression which matches to the *Upper_expression*, and the resulting expression is obtained by replacing the subexpression with the appropriate expression that is represented by the *Lower_expression*. An EUODHILOS system automatically generates all the possible forms of an expression which may be obtained by successive applications of a given rewriting rule. Users can then choose an intended one from them. For considering that some rewriting rule may cause infinite applications, the maximum number for applications are set in the system.

Like an inference rule, a rewriting rule may have some syntax category specifications on their expressions.

(4) Consistency Checking

Similarly to the consistency checking between language definitions and the data for logical systems described in Section 4.3, the consistency checking between derivation system definition data and proof data is also crucially important for G-RAS systems like EUODHILOS. This is because the once valid proofs of theorems, derivation systems and other proof fragments may become invalid as one or more axioms or inference rules have been changed. For supporting such consistency checking, EUODHILOS is supposed to be equipped with the facility of automatic correction of the proof data in order to maintain the renaming of axioms and inference rules.

(5) Dependency Calculation Method

The dependency status of a conclusion is automatically calculated by the ordinary way of natural deduction[89]. Other dependency calculation also can be dealt with in EUODHILOS if we specify it by using an idea of dependency as a tag/label together with the rewriting rules. For example, let us consider the $\wedge$-introduction rule of some relevant logic,

$$\frac{A^{\alpha} \quad B^{\alpha}}{A \wedge B^{\alpha}}$$

where the superscript α denotes the dependency on which the formula depends. The rule says that we can infer the formula $A \wedge B$ with dependency α only if we have

A and B with the same dependency α. Such a rule may be very naturally specified within the rule description convention of EUODHILOS by incorporating dependency into an object formula, as follows:

$$\frac{\alpha \Rightarrow A \quad \alpha \Rightarrow B}{\alpha \Rightarrow A \wedge B}$$

There some possibilities on how the tag part is considered to be identical. We can describe such conditions in terms of rewriting rules.

5.2 Side Condition

EUODHILOS allows users to attach one or more side conditions to inference rules and axioms. A side condition consists of an arbitrary number of combinations of five primitive side conditions, `FREE-FOR`, `NOT-FREE`, `NOT-FREE-IN-ASSM`, `FULL-SUBST`, and `SYNTAX-CAT`.

5.2.1 Primitive Side Conditions

(i) `FREE-FOR` (Substitution Condition)
This one specifies the substitution condition. "/" indicates the actual substitution so that in order to specify a substitution, this side condition will be needed most of the situations. For example, the Hilbert style axiom

$$\forall x.A(x) \supset A(t/x)$$

of the predicate logic needs the side condition that says, the variable "x" in the formula "A" is substitutable only if no free variables in the term "t" are newly bounded in the substitution. This condition is described as:

```
(FREE-FOR ("t" . Term) ("x" . Variable) ("A" . Formula)).
```

(ii) `NOT-FREE` (Variable Occurrence Condition)
This is the condition that specifies that the given variable does not occur freely in a given formula. For example, the axiom

$$\forall x.(A \supset B(x)) \supset (A \supset \forall x.B(x))$$

of the predicate logic is valid if the variable "x" does not occur freely in the formula "A". This condition is described as:

```
(NOT-FREE ("x" . Variable) ("A" . Formula)).
```

(iii) `NOT-FREE-IN-ASSM` (Eigenvariable Condition)
This condition can be attached only on the inference rules. It is the condition for eigenvariables in the natural deduction formalism. For example, the inference rule:

$$\frac{A(y/x)}{\forall x.A(x)}\ \forall I$$

has the side condition:

```
(NOT-FREE-IN-ASSM ("y" . Variable) ("A(y/x)" . Formula)),
```

which states that the "`Variable`" "y" must not appear freely in any assumptions upon which the "`Formula`" "`A(y/x)`" depends.

Note that the eigenvariable condition for Gentzen's sequent calculus can be specified by using the variable occurrence condition described in the previous item.

(iv) `FULL-SUBST` (Full substitution)
This condition specifies that the substitution expression that appears in an axiom or an inference rule denotes the full substitution. For example, in the example inference rule in the previous item, the rule must also have the side condition:

```
(FULL-SUBST ("A(y/x)" . Formula)),
```

because the substitution must be applied to all occurrences of "x".

(v) `SYNTAX-CAT` (Syntax Constraint)
This condition specifies a syntax category to a metavariable. For example, in the sequent calculus, the identity axiom:

$$A \vdash A$$

may be given the restriction:

```
(SYNTAX-CAT ("A" . AtomicFormula)),
```

which states that the formula represented as "`A`" must have the syntax category "`AtomicFormula`", which is the nonterminal for the class of atomic formulas.

5.2.2 Satisfiability Checking of Side Conditions

Among the side conditions mentioned in the previous section, the syntax constraint condition can be checked instantly from the internal representation of the logical formula. For other side conditions, the concept of bound variable and its scope is essentially important. When the parse data for syntax analysis are generated from the user-defined syntax definition, the procedure also generates the data for dealing with bound variable and its scope in an internal representation of a formula.

For example, from the production rule:

```
Formula ::= "∀ " @Variable "." [ Formula ];
```

the list data such as:

```
((Formula "∀ " Variable "." Formula) 1 3)
```

will be generated. The first element is the S-expression of the production rule which is mentioned in Section 4.3. The symbols for denoting the bound variables and their scopes are removed. The second element denotes the bound variable and the rest elements indicate the position of its scopes. The position number starts from 0 for the first element among the child-nodes. In this example "∀" is the 0th element, the string representing the `Variable` item is the bound variable, and the string for `Formula` is considered its scope. The verification of side conditions in terms of occurrences of variables will be processed recursively from the root node of the internal representation of the logical formula. When a production rule that includes a bound variable appears in the verification, the verifier program checks the above data and when the scope is checked it supposes that the corresponding variable is already bounded, and continues the checking process.

5.3 Chapter Summary

In this chapter we have described the framework for derivation system specification. The aim of the derivation system is to specify the deductive structure of the target domain. Flexibility of specification styles is one of the most important feature in order to adjust itself to the user's preference. The derivation system of EUODHILOS consists of three components: axioms, inference rules, and rewriting rules, which are based on the natural deduction style formalism. These rules are given and modified in tree-form so that they are easily recognized by the users.

An axiom is generally represented in a schematic expression with its name. An inference rule consists of the conclusion, one or more premises and optional assumptions together with optional combination of primitive side conditions. A rewriting rule consists of an upper and a lower expressions that specifies the expressions before and after rewriting.

Users may choose any combination of these components so that he or she can formulate the structure of the target domain according to its structural character and his or her preference or philosophy. This framework is general enough for dealing with a wide variety of types of derivations. It is completely up to the user which type to choose.

Chapter 6

Proof Construction

The major drawback of reasoning in formal logic is that derivations tend to be lengthy and tedious because the detailed level of derivations are required in reasoning. Furthermore, performing formal derivations is time-consuming and error-prone. We notice that such a situation is quite similar to the formal development of programs in which programs are derived or transformed and properties of programs are established. The use of computers for formal reasoning can overcome the problems about such errors and the time-consuming task. EUODHILOS systems aim to provide users with the facilities which are able to support natural and efficient proof constructions in the defined formal systems[60; 62].

The aim of this chapter is to realize a proof assisting environment that is natural and efficient. In order to achieve this aim, it is quite important for the system to adapt to the user's reasoning style. It is also important to provide a user-interface where the user is able to recognize the reasoning situation easily and correctly.

We investigate such a proof construction supporting environment in two issues. The first issue is the environment where users are easy to recognize the situations and the facilities for helping flexible reasoning. The second issue is the proof editing, or manipulating, functions. With a variety of functions that match the user's reasoning style, he or she will feel comfortable to find appropriate proofs of the problems at hand.

This chapter is organized as follows. In Section 6.1, we take out some features that are important for assisting flexible proof construction. In Section 6.2, we present the proof manipulation features. These are also important for flexible reasoning in the actual proof construction. In Section 6.3, we are concerned with the automated reasoning with the use of tactics. In Section 6.4, we conclude this chapter.

6.1 Flexible Proof Constructions with Sheet of Thought

The most important facility for assisting proof construction is to provide a comfortable proving environment on which the users easily recognize where they are in the reasoning and are able to control the whole reasoning processes according to their own styles. The proof assisting environment of EUODHILOS is called the "sheet of thought". In the rest of this section we demonstrate the importance and usefulness of the facilities of the sheet of thought in interactive reasoning by discussing its features from several different aspects.

6.1.1 Proof Construction Style

The proof constructions on the sheets of thought are basically performed through the interactions between the system and the users. In each step of rule application procedures, users are supposed to denote one or more proof fragments as the target proof fragments, the rule to be applied, the place to apply, and derivation direction. If more than one results are obtained as the result of the application, the system generates all the possible results, displays them, and waits for the user to decide which one to choose. By taking such a method, the users are able to avoid a large amount of various extra operations to be demanded in derivations. There is another advantage in this method that errors in inputting will be vigorously reduced.

The concept and the term "sheet of thought" originated from a metaphor of work or calculation sheet and is apparently analogous to the concept of sheet of assertion which is originated to C. S. Peirce. He actually developed an extensive diagrammatic calculus which he intended as a general reasoning tool. A sheet of thought, in our case, is a field of thought where we are allowed to draft a proof, to compose proof fragments or detach a proof, to reason using lemmas, etc., while a sheet of assertion is a field of thought where an existential graph as an icon of thought is supposed to be drawn. Proof construction by the use of sheets of thought turns out to yield proof modularization, which is considered important particularly for proving in the large scales. It may be beneficial to note that proof modularization is approximately equal to the concept of program modularization, to borrow the term from software engineering.

6.1.2 Tree-form Proof Representation

In order to deal with a wide variety of logical structures in a uniform fashion, the EUODHILOS system supports the proof construction style in such a way that the procedures are controlled by the user. To be precise, the construction of proofs on a sheet of thought starts with putting axioms and assumptions, or goals on the sheet. By applying the rules of the logical system, the small proof fragments grow bigger and bigger until eventually the complete proof is constructed. It's basically up to the users what actions to take in the proof processes. Obviously from this interaction model of the system, it is quite important for the system to display proofs in the representation that the user can capture the proof structure with ease. In order to meet such the requests, the inference and the rewriting rules of EUODHILOS are displayed in a natural deduction style as was presented in Chapter 5. This naturally induces that the organization of a proof on a sheet of thought is represented in tree-form with a justification for each node indicated in the right margin.

For example, the justification part ($\supset$E{1,2}) of the following proof tree says that the conclusion formula B is obtained by applying the rule named $\supset$E from the two proof fragments, of which the conclusion formulas turn to be used as premises A and A$\supset$B, respectively, and the conclusion formula B depends on the assumptions with the identification numbers 1 and 2 that might be located above the premises.

$$\frac{\begin{matrix}\vdots & \vdots \\ A & A \supset B\end{matrix}}{B}\ (\supset E\{1,2\})$$

Consequently it leads to the explicit representation of a proof structure, in other words, proof visualization.

Displaying the proofs in a tree-form is really good for those proofs that have sufficiently small display sizes. For fairly large proofs that the whole proof trees can not be displayed in one screen, we have two ways to deal with them. One is to reduce the size of a proof that is too big to display in a screen. EUODHILOS provides the facility to register reusable proofs as theorems and derived rules. By using these, a large proof can be represented in smaller size. This method makes sense if the large proof can be recomposed with such small components. This is also good for the user to understand the structure of the proof. Even if we are able to display quite a large proof tree, it will be quite difficult also to understand the complicated large tree. Thus it is better to reduce the proof size so that it becomes easy to read and understand how it is constructed.

However, sometimes it is quite difficult to apply this type of size reduction method. The second method to reduce the display size is to display a proof in an abridged form, where only the important parts are displayed. For this purpose, EUODHILOS-II provides two ways of representation with full tree form and abridged form so that users can choose either form to take in reasoning according to their preference(See Figure 3.4).

Suppose a user applies a derivation rule to a proof fragment. In such a case the most important information is the part where the rule is applied. Rest parts of the proof have not essential importance. For forward derivation, the conclusion part of a proof, and at best the effective assumptions, i.e. the assumptions that are not yet discharged, are important. For backward derivation, the effective assumptions are the only essential information for performing the derivation. Take, for example, the following proof fragment in order to forwardly apply a rule:

$$\frac{\dfrac{[A]^1 \quad [B]^2}{A \wedge B}}{A \supset A \wedge B}$$

(where $[A]^1$ has been discharged,) only the conclusion formula $A \supset A \wedge B$ and the assumption B that the formula is depending on are important. For backward derivation, only B is important. However, if the system displays only the dependency relations for assumptions, the user cannot get the information which derivation rule is applied for getting the result. Then he or she may apply introduction rules and elimination rules one after another so that redundant proofs may be constructed without recognizing it. So in EUODHILOS-II the standard representation form of a proof is to show the conclusion, premises and assumptions the conclusion depends, and the rule that deduces the conclusion.

On the other hand, if we want to separate a proof at a formula so that they can be used in other proofs or used as one or two lemmas, the whole proof tree should be displayed to the user. In this way, he or she can recognize the target formula in the proof. In such a case, users can change the proof representation mode and operate on the fully-displayed proof fragment.

It will be worth noting that the resulting proofs can be saved as a LaTeX macro so that the proof structures can be viewed with a DVI previewer, as shown in the examples in Figure 7.2 and Figure 7.8 in Chapter 7.

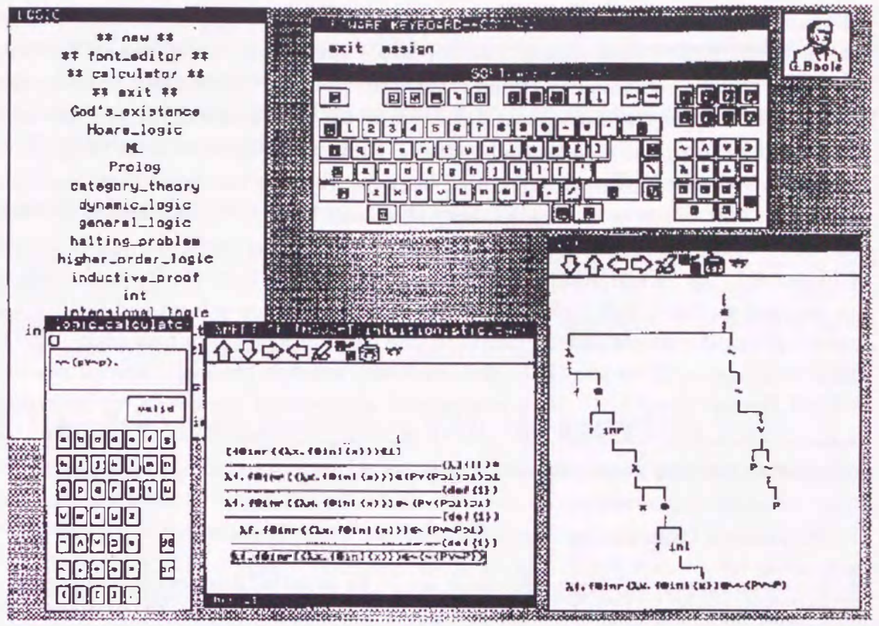


Figure 6.1: Reasoning-Oriented Human-Computer Interface

6.1.3 Inputting Characters and Expressions

(i) Software Keyboard and Font Editor

In our formal systems, particularly in logical systems, we often want to use special symbols in reasoning. For example in typical logics, we need the symbols like "$\supset$", "$\wedge$", "$\forall$", "$\exists$", etc. and etc. Most reasoning systems are able to deal with only ASCII characters. Thus the logical symbols "$\rightarrow$" and "$\supset$" are often substituted with "->", and "$\vee$" are substituted with "\/" or "v", so that all the expressions are represented in some combinations of ASCII character strings. However it is quite difficult to replace for these symbols like "$\equiv$", "$\simeq$", "$\in$", "$\propto$" and so on. This is a reason for the users to have difficulty in capturing logical expressions in a natural and easy to understandable way. In some systems such as ωple[19] and PROOF DESIGNER[5],

users are allowed to input the natural logical symbols. According to the basic policy, also in EUODHILOS, the system has to provide the means to define new symbols and to use them in reasoning. To fulfill this requirement EUODHILOS-I, provides the software keyboard for inputting user-defined symbols. The shape of the new symbols are defined by using the standard font editor. They are assigned to some of the keytops of the software keyboard. By typing the keys, the user can input these symbols and thus is able to use in the syntax description and thus in the formulas of axioms, derivation rules, and proofs. A software keyboard appears in in the upper center of Figure 6.1. In EUODHILOS-II, logical symbols and a part of Greek characters are assigned to ISO Latin 1 character area. The system has a front-end facility for inputting logical symbols and the users are able to input logical symbols easily. EUODHILOS-II also runs on the GNU Mule systems, which is the multi-lingual version of GNU Emacs. Since GNU Mule is equipped with several front-ends for inputting Japanese characters, users are also able to input logical symbols as the similar way to inputting ordinary Kanji characters easily.

(ii) Keyboard Operations

In the reasoning process, logical formulas are to be inputted frequently, and furthermore these inputting operations are performed through keyboards, EUODHILOS-II system is designed to support inputting proof editing commands also by using the keyboard. For example, the command for moving the cursor to the next item is assigned to the key "n" (next) in every editing environment in various situations. The terminating command is, similarly, assigned to "q" (quit). Users can check all the available commands and their assignments with the help window whenever he or she wants. The syntax definitions and side conditions are also able to be checked at any time. These facilities contribute to the user-friendly operations in EUODHILOS-II.

(iii) Inputting Logical Expressions

Derivations begin with putting any one of assumptions, premises, theorems and conclusions on sheets of thought. Axioms and theorems are inputted simply by selecting one at a time from the axiom list and theorem database respectively. Users also input any formula on a sheet of thought as an "assumption." The formula may be used, actually, either as an assumption or as a goal of a proof depending on how it is used in the first derivation, which is determined by the user. In the former case, the "assumption" is treated as, literally, an assumption and it is used in a forward

derivation. In the latter case, the "assumption" is used as a conjecture or a goal in a backward derivation. The inputted formula is checked by the system whether it is an axiom or a theorem. If an "assumption" is either one of them it will be treated as an axiom or a theorem depending on what it is. This checking may be done for any assumption of any proof fragment.

(iv) Formula Editor

This is a structure editor for logical formulas and makes it easy to input, modify and display complicated formulas. In addition to ordinary editing functions, it provides some proper functions for formulas such as rewriting functions. The window at the right down corner of Figure 6.1 is the formula editor.

6.1.4 Schematic Reasoning with Metavariables

Metavariables play an important role in logic specification and reasoning. A metavariable is a place-holder which can be substituted with any expression of the same syntax type. Such a substitution is called an "instantiation".

EUODHILOS is supposed to make the meta and object distinction at the time of language definition. Then substitution and unification viewed as the common and primitive symbol operations are supposed to operate on metavariables, in addition to the usual variables.

In the logic specification step, metavariables are used to make schematic expressions, including schematic formulas. Take, for example, the axiom "$0 = 0$". This is concerned only with the specific object "0". It says nothing about "1" nor about any other objects. On the other hand, if we use a metavariable, say "x", we can write "$x = x$", which asserts the identity relation for all objects, including "0" and "1".

The metavariables are useful not only for the schematic specifications of axioms, inference rules or rewriting rules, but also for schematic proofs. Thanks to the metavariables we can start reasoning without specifying in detail. If we do not have a specific expression in advance, we just put an appropriate metavariable and start reasoning.

In EUODHILOS, we can instantiate the metavariable when the specific expression becomes clear. Some instantiations occur automatically; for example, when two proof fragments are connected. Suppose we have two proof fragments. We specify the conclusion of one of them and also specify a premise of the other. Then the system checks if the two formulas are unifiable by instantiating these proof fragments appropriately.

If possible, the system calculates the most general unifier(mgu) automatically and connecting them by instantiating with the mgu. This is also a useful construction method for making a large proof.

A schematic proof, however, sometimes need to be careful since the metavariables in the proof may not be fully instantiated so as to promote further steps. For example, the first-order axiom: $\forall x.(P \supset Q) \supset (P \supset \forall x.Q)$ has the condition that x is not free in P, where P and Q, and x are metavariables ranging over formulas and individual variables respectively. Note that such a condition in axioms may be viewed as a side condition like those of inference rules. Thus a proof using this axiom turns out to be schematized to the extent that the metavariables P are instantiated as concrete formulas. In the case of inference rules, a proof process may be banned by its side conditions unless metavariables are sufficiently instantiated so as to be able to check side conditions. An alternative to handle these situations would be to delay checking side conditions until metavariables are fully instantiated. To do so, every side condition which has been inherited as unchecked during the proof process would have to be kept with the final theorem which is not actually a theorem, but should be stored as a conditional theorem. EUODHILOS takes this style for side conditions.

6.1.5 Candidate Generation

It may happen that more than one proofs are obtained in an application of a rule. For example, if we take the proof

$$\frac{\Pi}{B(t,t)}$$

and apply the inference rule

$$\frac{A(t/x)}{\exists x.A(x)}\ \exists I$$

forwardly, the following results are obtained because $A(t/x)$ represents generic partial substitution, according to the occurrence of the variable x.

$$\frac{\frac{\Pi}{B(t,t)}}{\exists x.B(x,x)} \quad \frac{\frac{\Pi}{B(t,t)}}{\exists x.B(t,x)} \quad \frac{\frac{\Pi}{B(t,t)}}{\exists x.B(x,t)} \quad \frac{\frac{\Pi}{B(t,t)}}{\exists x.B(t,t)}$$

In order to choose one of them, the other proof assisting systems take the methods such as specifying the position of the variable where substitution occurs or giving the formula that is before the substitution. However if the target formula is complex in some sense, the specification becomes very complex and it becomes error-prone. EUODHILOS presents all the possible conclusions and the candidates formulas that are to be substituted, and the user chooses the appropriate one among them. This function reduces a lot of mistakes of the user's inputting and it contributes to efficient reasoning.

6.1.6 Reusing of Proofs

It is highly preferable to enable the user to reuse proofs in the later proving processes. Often we recognize a pattern of proofs that appears again and again. Once these proofs are stored in the theory database, they can be used as if they were obtained in one step like primitive inference rules and axioms. They are referred to and reused in the later proofs for other theorem proving processes. For large and complex proofs, they are helpful for preventing from generating proof trees that are too big to deal with easily, and avoiding the repetitive occurrences of the same subtree in a proof tree. This enables us not only to reduce the size of the proof but also to grasp the proof structure as a whole without being concerned with the details.

EUODHILOS has two kinds of reusable data: theorems and derived rules. If the conclusion of a proof fragment does not depend on any assumptions, it means that the conclusion formula holds without any assumptions, or stand-alone formula. Then it will be registered as a theorem and can be used like an axiom. On the other case, if the conclusion depends on one or more assumptions, the proof fragment as a whole will be saved as a derived rule. The assumptions turn out to be the premises of the derived rule, and the conclusion of the proof fragment turns out to be the conclusion of the rule. A derived rule can be used like an inference rule.

To summarize, the use of theorems/lemmas and derived-rules in reasoning has the following advantages:

(i) By packing an amount of proving steps the whole proof looks in a reasonable size so that it is easier to make big proofs than without them.

(ii) Such a feature gives a name to one package of proof so that users can understand the proof easily.

(iii) It is rare that we have a complete plan of a proof in advance, thus most proofs are made in a trial-and-error fashion. A proof starting in a rough plan may fail after some number of proving steps. Even in such a case, one or more parts of the partly-constructed proof fragments may be used as parts of another proofs. Some relatively small proofs may be used so that we can construct a big proof by combining these useful proof fragments.

A facility for expanding theorems and derived rules into original proofs is also useful. Users can modify parts of the proof and reuse them in other proofs. This is a facility for learning from previous proofs.

It would be quite usual to take much time for a proof to be completed, in particular for a large and complex proof. EUODHILOS has not only a theorem database but also a work area for temporarily storing various proof fragments on sheets of thought which may be or may not be useful. Some of such small pieces of proof fragments are found to be useful in later reasoning. So such a facility is also preferable in G-RAS systems.

6.2 Proof Manipulation

One of the notable features of human reasoning is its varieties of reasoning styles. Humans combine various reasoning styles freely according to the reasoning situations. Sometimes, they derive conclusions from assumptions (i.e. forward reasoning), while in other times, they deduce sub-goals from a goal that is to be deduced in reasoning (i.e. backward reasoning). They often combine these styles mixed up or deduce both sub-goals (or sub-premises) and new conclusions from a same proposition. It also happens that they fill in the gap between an assumption and a conclusion and obtain a connected big proof.

In this section we discuss various styles of human reasoning with presenting how they are handled in EUODHILOS.

6.2.1 Forward Reasoning

In order to deduce forwardly by applying an inference rule, we usually start a proof by inputting formulas used as premises of the rule and in a natural deduction setting by further indicating assumptions to be discharged. Then we may select an appropriate inference rule from the rule menu which has been automatically generated at the time of logic definition, or we may input a formula as the conclusion. If we select a rule, the

system applies the rule to the premises and assumptions, and derives the conclusion. If we give the conclusion, then the system searches for the rules and tries to find one which coincides with this deduction. EUODHILOS searches for the candidates of applicable inference rules to the given premises as well and hence we can simply choose the intended one.

6.2.2 Backward Reasoning

Backward reasoning starts with specifying a goal, which is a proof fragment consisting of a single formula that has not been proved to be a theorem. In backward reasoning, a rule application to a goal formula deduces one or more subgoals to be proved. Backward reasoning terminates when all the subgoals are proved to be axioms or theorems, or are to be assumptions that are discharged in rule applications.

In a backward reasoning, the intermediate proof may branch off to partially justified proof fragments and the complete justification of those partially justified proof fragments is delayed until the completion of a final proof tree.

6.2.3 Proof Connection and Separation

From our observation on human reasoning, we recognize that we humans often use both forward and backward reasoning. To support human reasoning, therefore, the system must provide a mixed reasoning feature. Fortunately, mixed reasoning in EUODHILOS is fairly easy to achieve because the user can change the deduction direction modes from one to the other at any time in the reasoning process. Depending on whether the system is in the forward or backward mode, the rules are applied in the forward or backward method respectively.

In natural deduction setting of a formal system, forward reasoning may be advanced without inputting or indicating assumptions to be discharged. This implies that at an appropriate stage of a proof, we have to decide which assumptions should be discharged. This comes from such a proper form of an inference rule that assumptions in natural deduction rules may not be used in the derivations of premises. EUODHILOS helps us with doing such a task in a natural way. Let us consider the proof composition from the following two proof trees.

$$\begin{array}{c} [P]^1 \cdots [P]^2 \\ \vdots \\ \hline Q \end{array} (\ldots\{1,2\})$$

$$\frac{[Q]^3}{P \supset Q}\,(\supset I\{3\})$$

By simply composing them, we get

$$\frac{\begin{array}{c}[P]^1 \cdots [P]^2 \\ \vdots \\ Q\end{array}}{P \supset Q}\,(\supset I\{1,2\}).$$

We call such a manipulation a "connection".

In such an operation the proposition P in the consequence is not known whether it is P^1, P^2, or any other P outside this proof tree. EUODHILOS supports the following discharging method:

(a) If the proposition P in the consequence is meant to be $P^1(P^2)$, then we choose $P^1(P^2)$ as a discharged assumption and get the new proof tree with the new justification $(\supset I\{2\})((\supset I\{1\}))$ respectively.

(b) If the proposition P in the consequence is meant to be any other P outside the proof tree, then we may simply continue expanding the proof without any further action.

To connect two proof fragments in EUODHILOS, we specify the conclusion part of one proof fragment and one of the assumptions of another fragment. Then the system automatically finds the proper substitution for the metavariables in these proof fragments and connects them if possible. In what follows we summarize various styles of reasoning of EUODHILOS in more detail.

(i) Connection by complete matching

Two proof fragments can be connected through a common formula occurring in them when one of them is a hypothesis and the other a conclusion. The process begins with selecting the two formulas and invoking the proper operations. As the result, the proof fragments are connected into one proof fragment. Schematically, this amounts to attaining the following inference figure which is viewed as one of Tarski's consequence relation common in all logics.

$$\frac{\Gamma \vdash C(\text{on a sheet of thought}) \quad \Delta C \Sigma \vdash A(\text{on a sheet of thought})}{\Delta \Gamma \Sigma \vdash A(\text{on a sheet of thought})}$$

where Γ, Δ and Σ might represent sequences of formulas (possibly empty), and A and C denote formulas in some defined logical system.

(ii) Connection by the use of a rule of inference

This is essentially a forward reasoning and may be called a distributed forward reasoning. The process is similar to the above except that the connection is done from proof fragments scattered on several sheets of thought through an appropriate rule of inference. Let us take an example schema of modus ponens:

$$\frac{\Gamma \vdash A \supset B(\text{on a sheet of thought}) \quad \Delta \vdash A(\text{on a sheet of thought})}{\Gamma \Delta \vdash B(\text{on a sheet of thought})}$$

with the same proviso that B represents a formula.

(iii) Connection with unification

Two proof fragments can be connected through two unifiable formulas occurring in them when one of them is a hypothesis and the other a conclusion. The process begins by selecting the two formulas and invoking the proper operations. As the result, the proof fragments are unified to the most general proof fragment. It is, however, noted that the unification can be done through metavariables.

(iv) Separation

The separation is the converse operation to the connection. The separation process begins with selecting a formula occurring in a sheet of thought and invoking the proper operations. As the result, the proof fragment is detached into two fragments. Separation is useful when a useful subproof structure is often found in a big proof. After obtaining the useful part, it can be saved as a derived rule and used in later reasoning.

In natural deduction setting of a logical system, the assumption numbers are automatically managed by the system. We will illustrate this by separating the following proof tree at the location of formula B.

$$\frac{\dfrac{\dfrac{[A]^1 \quad [A \supset B]^2}{B}\,(\supset I\{1,2\})}{C}\,(\ldots\{1,2\}) \qquad [D]^3}{E}\,(\ldots\{1,2,3\})$$

We then get the two proof trees on a sheet of thought as follows.

$$\frac{[A]^1 \quad [A \supset B]^2}{B}\,(\supset I\{1,2\}) \qquad \text{and} \qquad \frac{\dfrac{[B]^4}{C}\,(\ldots\{4\}) \quad [D]^3}{E}\,(\ldots\{3,4\})$$

6.2.4 Internal Proof Manipulation Procedure

In this section we describe how the actual proof manipulations are performed inside of the system. Since proofs are represented according to the natural deduction style, the representation and manipulation procedure are rather complicated than other types of representations based on the Hilbert style and sequent style.

(i) Internal Representation of Proofs

There are three styles of internal representations of proofs.

(a) Axioms and Theorems:

Axioms and theorems are internally represented with internal representation of its formula together with their identifiers and names.

```
((axiom   . Name) . Axiom)
((theorem . Name) . Theorem)
```

A theorem is a conclusion of a complete proof already obtained and registered as a theorem in a theory database.

(b) Assumption:

There are two types of assumptions; ordinary one and discharged one. The former is labeled with "`assumption`", and the latter is labeled with "`discharged`".

```
((assumption . Label) . Assumption)
((discharged . Label) . Assumption)
```

"Label" is the identifier(a natural number) that discriminates assumptions. Even the assumptions having literally identical formulas may have different labels. Then, they

are treated as different. On the other hand, the assumptions having the same label are treated as equal and will be discharged at the same time. Note that the discharged assumptions are generated only in a deduction that discharges assumptions. Therefore they only appear as subexpressions of the internal representations of derived proofs as will be described in the next item.

(c) Derived Proof:

This represents a proof that is obtained by applying a derivation rule to one or more proofs.

```
(Conclusion .
  ((Name  Side_Condition  Label_of_the_discharged_assumptions) .
                                                  (Proof_1 ... Proof_n)))
```

Here, the "Name" indicates the name of the derivation rule, "Side Condition" is the list of side conditions that are obtained by instantiating those of the derivation rule by substituting in the applications of the rule. Checking of the side conditions can be made at any time in the midst of a proof. In such a case, each side condition of each derivation step is checked. These side conditions are instantiated when the proof is instantiated. Thus the side conditions in a proof can be checked as it is. The "Label of the discharged assumptions" is the list of the labels that are attached to the assumptions and are discharged in the application of the derivation rule. This part will be referred to when a proof separation is made. The identifier "`discharged`" that is attached to the target assumption will be back to the identifier "`assumption`." A "Proof" is the internal representation of the proof that is a premise of the derivation rule.

The internal representation of a proof consists of the data that represent the overall proof structure and a representation that is called the proof status. A proof status has the information that corresponds to the abridged representation of a proof structure that was mentioned in the previous section. This is the data that is referred to when the system accepts an editing command from the user and when the system displays an abridged proof in each step of the proof construction. Such a data is every time generated from the internal representation of the proof data whenever a new derivation is performed. The format is similar to the internal representation of an inference rule:

$$
\begin{array}{l}
(\mathit{Conclusion}\ . \\
\quad (\mathit{Rule\ Name}\ . \\
\qquad\qquad ((\mathit{Premise}_1 \quad \mathit{Assumption}_{1,1} \ldots \mathit{Assumption}_{1,n(1)}) \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad (\mathit{Premise}_m \quad \mathit{Assumption}_{m,1} \ldots \mathit{Assumption}_{m,n(m)}) \ )))
\end{array}
$$

Where "Conclusion" is the conclusion formula of the proof, "Rule name" is the name of the rule that deduce the conclusion, a "Premise" is a formula that is a premise of the rule. "Assumption"s are the assumptions that each premises depends on, including the discharged ones. All the formulas are represented in the internal representation form as was described in Section 4.3.

(ii) Derivation Procedure

The derivation procedure varies depending on the derivation direction(i.e. forward or backward) and the type of the applied rule (i.e. inference, derived, or rewriting). In this section, we describe how they are processed by presenting the algorithm for forward derivation of inference and derived rules.

Step 1:

First, the user specifies the proof fragment that is the target of the derivation rule, the target place of the application of the proof fragment, and the derivation rule to be applied. The target place is internally represented as a list of the lists of premises and the list of assumptions to be discharged (called the "assumption list of proof fragment"). Suppose, for example, we have chosen two proof fragments:

$$
\begin{array}{c}
[V][W] \\
\vdots \\
\hline
U
\end{array}
\qquad\qquad
\begin{array}{c}
[Y] \\
\vdots \\
\hline
X
\end{array}
$$

where, we have also chosen the assumptions V and W of the former and Y of the latter to be discharged. Then the assumption list of these proof fragments is:

$$((U\ V\ W)\ (X\ Y))$$

Step 2:

Produce a list (called the "assumption list of derivation rule") of the element that consists of premises and the assumptions to be discharged from the definition data of the given derivation rule. For example, the assumption list of the derivation rule

$$
\begin{array}{c}
[B][C] \quad [E] \\
\vdots \qquad \vdots \\
\hline
A \qquad D \\
\hline
F
\end{array}
\qquad \text{is} \qquad ((A\ B\ C)\ (D\ E)).
$$

Each element corresponds to that of the assumption list of a proof fragment as was described in the previous step.

Step 3:

Generate the list (called the "assumption permutation list") that consists of all the permutations of the assumption list of the derivation rule that retains the relationship between a premise and its assumptions. For example, the assumption permutation list of the example of the previous step is:

$$
((A\ B\ C)\ (D\ E)) \Rightarrow
\begin{array}{l}
(\ ((A\ B\ C)\ (D\ E)) \\
\quad ((A\ C\ B)\ (D\ E)) \\
\quad ((D\ E)\ (A\ B\ C)) \\
\quad ((D\ E)\ (A\ C\ B))\ )
\end{array}
.
$$

This process is needed so that the system applies the derivation rule also to the following cases:

$$
\begin{array}{c}
[C] \quad [B] \quad [E] \\
\vdots \qquad \vdots \\
\hline
A \qquad D \\
\hline
F
\end{array}
\qquad \text{and} \qquad
\begin{array}{c}
[E] \quad [B] \quad [C] \\
\vdots \qquad \vdots \\
\hline
D \qquad A \\
\hline
F
\end{array}
,
$$

where premises and their assumptions may be permuted.

Step 4:

Leave the list elements of the assumption permutation list that have the same length as the specified assumption list of the proof fragment. This is a filtering process to eliminate the candidates that have different numbers of either premises or assumptions. In the current example, the assumption list of the proof fragment is

$((U\ V\ W)\ (X\ Y))$. Thus only the candidates that have the elements with lengths 3 and 2 are left.

$$\begin{array}{ll} (& ((A\ B\ C)\ (D\ E)) \\ & ((A\ C\ B)\ (D\ E)) \quad) \end{array}$$

is the result of this step. If the resulting list is empty, it means this derivation rule is not applicable to these proof fragments and thus ends up the whole procedure.

Step 5:

Firstly the system tries to make a matching to each of the elements obtained in the previous step against the assumption list of the specified proof fragments and creates the instantiation list that describes how to instantiate the metavariables. An instantiation list is represented as:

$$\{p_1 \leftarrow e_1, \ldots, p_n \leftarrow e_n\}.$$

This says that the metavariables represented in the pattern p_i is to be instantiated with e_i. If all the matchings fail and the returned list is empty, then the deduction procedure ends just like the fourth step.

Step 6:

If one or more metavariables are attached with substitution expressions in the derivation rule, the system checks the consistency and completions of the substitutions obtained in Step 5. The candidates that have the inconsistent substitutions are eliminated. For example,

$$\{A(x) \leftarrow B(y),\ A(t/x) \leftarrow B(b),\ x \leftarrow y,\ t \leftarrow a\}$$

is inconsistent. The expression $A(x)$ denotes the expression A where the occurrences of x are focused. Thus the expressions $A(x)$ and $A(y)$ are essentially the same as a formula pattern except their focuses are different. Note that if these expressions appear in the right side of "$\leftarrow$", they are treated as different. In such a case this type of expression indicates the result of the substitution to the specified occurrences.

The pattern $A(t/x)$, on the other hand, indicates the pattern where some of the occurrences of x in $A(x)$ are substituted with t. Therefore, an instantiation of $A(t/x)$ will be made by substituting to $B(y)$, which has been obtained from $A(x)$,

with replacing some y (the instance of x) with a (the instance of t). Thus only $B(y)$ or $B(a)$ are only allowable ones but it is against the substitution expression $A(t/x) \leftarrow B(b)$.

In another example,

$$\{A(t/x) \leftarrow B(a),\ x \leftarrow y,\ t \leftarrow a\}$$

an instantiation to the substitution expression $A(t/x)$ exists. If it is impossible to find an instantiation of $A(x)$, the system will calculate and make a completed substitution. This is the completion procedure.

In this example, in a similar reason to the above example, the instance of $A(x)$ is either $B(y)$ or $B(a)$. Thus the incomplete instantiation should be replaced with two instantiations:

$$\{A(x) \leftarrow B(y),\ A(t/x) \leftarrow B(a),\ x \leftarrow y,\ t \leftarrow a\}$$

$$\{A(x) \leftarrow B(a),\ A(t/x) \leftarrow B(a),\ x \leftarrow y,\ t \leftarrow a\}$$

If all the instantiations are inconsistent and have been removed from the list the whole procedure ends in this step.

Step 7:

Some metavariables that appear in the conclusion formula of the derivation rule may not appear in the instantiation list. The system will ask how to instantiate for each metavariable that is not specified yet. The resulting instantiation information is added to the instantiation list.

Step 8:

Apply the instantiation operations of the list obtained in the previous step to the side conditions of the derivation rule. Only the proof fragments that satisfy the side conditions are left as candidates. Therefore, as in other steps, if the resulting candidates list is empty the derivation fails in this step.

Step 9:

Apply each instantiation of the instantiation list to the conclusion formula of the derivation rule. If more than one conclusion candidates are generated the system

will ask the user which to choose (Section 6.1.5). For example, if we apply the $\wedge$-introduction rule to

$$\frac{\Pi_1}{A} \quad \text{and} \quad \frac{\Pi_2}{B},$$

the system will ask the user to choose either one of the two:

$$\frac{\dfrac{\Pi_1}{A} \quad \dfrac{\Pi_2}{B}}{A \wedge B} \quad \text{and} \quad \frac{\dfrac{\Pi_2}{B} \quad \dfrac{\Pi_1}{A}}{B \wedge A}.$$

Step 10:

Discharge the appropriate assumptions of the given proof fragment, and create the new proof fragments by using the conclusion formula obtained in the previous step.

The procedure is similar to this one for inference and derived rules in the backward application. However, the target proof fragment is only one, and no eigenvariable checking will not be done. Therefore it is simpler than the forward derivation described above.

An application of a rewriting rule will be made to only one proof fragment just like that of backward derivation, the application place is limited to either conclusion or the assumptions that are not yet be discharged, and it is needless to consider the discharging of assumptions. Thus the application procedure only considers the rewriting of formulas thus all the considerations can be made locally. As the result it is much simpler to apply the rewriting rules than inference rules.

6.3 Automated Reasoning

The automated reasoning in EUODHILOS framework does not aim for proving completely automatically because its basic concept is to assist human reasoning with computers. Instead of such a direction EUODHILOS investigates rather a semi-automatic reasoning with proof strategies represented with tactics and tacticals.

6.3.1 Tactics and Tacticals

Tactics and tacticals provide the means to automate deductions. A tactic specifies the outline of a proving procedure. A tactic of EUODHILOS takes a list of proofs and

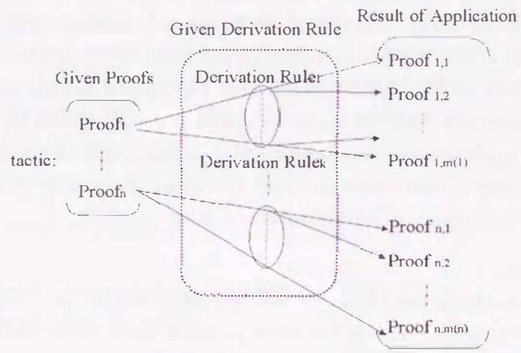


Figure 6.2: The Model of Tactic Application in EUODHILOS

generates the list of proofs obtained by applying the proof procedure described in the tactic. A tactical is a metafunction over tactics; it combines one or more tactics and creates a new one. Using tacticals we can create useful tactics that are complicated enough to express our intended proof procedures.

In EUODHILOS, tactics and tacticals are designed to describe the straightforward but tedious proof procedures. There are two reasons underlying this decision. One reason is that writing a large and generic tactic would either be too hard to describe accurately or inefficient. The other reason is that useful tactics, except those that are simple and straightforward, normally include ambiguities, so a lot of queries must be made to execute such a procedure, which will eliminate the advantage of using automated procedures.

A tactic of other prover systems like HOL[29] and Isabelle[85] takes a proof and a list of derivation rules, and parameters that specify which place to apply. It then returns the result of the application of the derivation rule that is found to be applicable in the first place. It is such a function in these systems. As is illustrated in Figure 6.2 a tactic in EUODHILOS, on the other hand, is a function which takes a list of proofs and the derivation rule as arguments and returns the list of all proofs that are obtained by applying the rule to the proofs given in the arguments. A list of derivation rules can be given as a regular expression of the names of the rules and such a pattern is considered to be a list of derivation rules those names match to the pattern. As

has been mentioned, the system allows the users to apply the derivation rules both forward and backward freely depending on the users intention. Therefore the tactics also deal with both forward and backward application of derivation rules, which is a characteristic feature of EUODHILOS tactics. Further, it is also characteristic that our system automatically collects premises that are not specified by the user but are necessary in the applications from among the axioms and theorems, and the user needs not to explicitly specify how to apply the given derivation rule.

The followings are some of the primitive tactics:

- `forward_tac`
  This tactic creates a list of proofs that are obtained in the forward applications of inference rules that match the name pattern given as an argument to each element of the proof list. If the inference rule discharges one or more assumptions, the tactic generates all the proofs that correspond to all the combinations of assumptions. Moreover, if the rule is an inference rule that requires more than one premises, the tactic fixes the proof fragments and tries to find other necessary proofs in the theorem, axiom, and proof fragments database as premises that can be used in the situation, and apply the rule to them.

- `backward_tac`
  For each element of the proof list, this tactic applies the inference rule in the backward manner whose names match to the pattern given as an argument, and generates the list of proof list. If there are more than one applicable places, one proof is generated from each place.

- `rewrite_tac`
  This tactic generates a list of proofs that are obtained by applying rewriting rules where the names match to the name pattern given in an argument to each proof of the given proof list. If there are more than one places that one of the rewriting rules is applicable, it applies the rule to each of them. Furthermore, if there are more than one expressions in an applicable formula, it also applies to each of the expression and generates the resulting proof. Note that a rewriting rule is supposed to be applied in both directions. Finally all the results are merged and become the final results.

A tactical is a functional that gives a combination of tactics. EUODHILOS is equipped with tacticals like appending lists(`append`), consecutive application(`then`), conditional choice(`orelse`), and repetition(`repeat`), like those of HOL and Isabelle.

`append` takes an arbitrary number of tactics as arguments and appends all the results obtained by applying the argument tactics one after another. `if`, `then`, and `orelse` are tacticals for conditional applications. `try` applies the tactic anyway. If the application fails, it keeps the proofs unchanged instead of returning the empty list. `repeat` repeats the applications of the argument tactic as many times as possible.

The following is an example of a tactic:

```
(deftac sample-tactic
  (then
   (repeat (forward_tac "∧E.*"))
   (forward_tac "⊃I")))
```

This tactic has the name `sample-tactic`. It applies the ∧-elimination rules expressed as rule name pattern "`∧E.*`" (i.e., "`∧E1`", "`∧E2`", "`∧E-left`", etc.) as many times as possible and then applies the ⊃-introduction rule in the forward direction.

6.3.2 Procedure of Tactic Application

The application procedure of proof tactics is essentially the same as that of interactive derivation rules. What are different in them are:

(i) In tactics, applicable derivation rules may be more than one, thus the proof fragments that are the target of the application may be more than one, thus all the results are merged finally, and

(ii) It automatically generates instantiations of metavariables that cannot be determined in the pattern matching in the application process.

We take up "`forward_tac`" for example and describe how it is applied.

Step 1:

Determine the set of proof fragments that are supposed to be used for the application of `forward_tac` and the set of derivation rules for application from an argument of `forward_tac`.

Step 2:

For each proof fragment and for derivation rule, it processes as follows:

Step 2.1: Check the length of the assumption list of the derivation rule and generates the list (called "proof pair list") that consist of the same number of elements, each of the elements is taken from axioms, theorems, and proof fragments that are given in the previous step. For example, if the derivation rule requires two premises, each element of the proof pair list consists of two elements. The first one is the specified proof fragment and the other is either one of the axioms, theorems, and proof fragments.

Step 2.2: Extract the assumptions that are not discharged yet from the elements of the proof pair list. Leave the assumptions that have the same number of assumptions to be applied by the derivation rule. For each proof fragment, generate the assumption list of the proof fragment that indicates where to apply the rule.

Step 2.3: Based on the data obtained in the previous step, apply the forward derivation algorithm described in Section 6.2.4. In order to perform the procedure more automatically, the system generates new metavariables for those metavariables that cannot be instantiated in the pattern matching. The newly generated metavariables are combined with prefix string given by the user followed by the automatically-generated numbers so that the name does not appear in the proof. All the results are returned.

Step 3:

Merge all the results and present them to the user as the candidates of the proof. The user chooses the one he or she wants as was described in Section 6.1.5.

For example "`(repeat (forward_tac "⊃ I"))`" is the tactic that says to apply the ⊃ introduction rule forwardly as many times as possible. If we apply this tactic to the proof fragment(1, 2, 3 are the labels for discriminating the assumptions):

$$\dfrac{\dfrac{[A]^1 \quad [A \supset B]^2}{B} \quad \dfrac{[A]^1 \quad [A \supset (B \supset C)]^3}{B \supset C}}{C}$$

$$\left[\begin{array}{cc}
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A]^1 \quad [A \supset B]^2}{B} \quad \dfrac{[A]^1 \quad [A \supset (B \supset C)]^3}{B \supset C}}{C}}{A \supset C}}{(A \supset B) \supset (A \supset C)}}{(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))}, &
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A]^1 \quad [A \supset B]^2}{B} \quad \dfrac{[A]^1 \quad [A \supset (B \supset C)]^3}{B \supset C}}{C}}{A \supset C}}{(A \supset (B \supset C)) \supset (A \supset C)}}{(A \supset B) \supset ((A \supset (B \supset C)) \supset (A \supset C))}, \\
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A]^1 \quad [A \supset B]^2}{B} \quad \dfrac{[A]^1 \quad [A \supset (B \supset C)]^3}{B \supset C}}{C}}{(A \supset B) \supset C}}{A \supset ((A \supset B) \supset C)}}{(A \supset (B \supset C)) \supset (A \supset ((A \supset B) \supset C))}, &
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A]^1 \quad [A \supset B]^2}{B} \quad \dfrac{[A]^1 \quad [A \supset (B \supset C)]^3}{B \supset C}}{C}}{(A \supset B) \supset C}}{(A \supset (B \supset C)) \supset ((A \supset B) \supset C)}}{A \supset ((A \supset (B \supset C)) \supset ((A \supset B) \supset C))}}{(A \supset B) \supset (A \supset ((A \supset (B \supset C)) \supset C))}, &
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A]^1 \quad [A \supset B]^2}{B} \quad \dfrac{[A]^1 \quad [A \supset (B \supset C)]^3}{B \supset C}}{C}}{(A \supset (B \supset C)) \supset C}}{A \supset ((A \supset (B \supset C)) \supset C)}}{(A \supset B) \supset ((A \supset (B \supset C)) \supset C)}}{A \supset ((A \supset B) \supset ((A \supset (B \supset C)) \supset C))}
\end{array}\right]$$

Figure 6.3: Resulting Proofs by Applying `forward_tac`

then we get the same number of proofs as the number of assumptions in one application of "`(forward_tac "⊃ I")`" and one assumption is discharged in the process. As the result, 6 (i.e. 3!) proofs are generated as the candidates of the results according to the order of discharging assumptions, which is illustrated in Figure 6.3.

6.4 Chapter Summary

In this chapter we have investigated the usable proof assisting environment and have discussed several issues that concerning this issue. Through these discussions we have demonstrated the usefulness of sheets of thought implemented in the EUODHILOS systems.

- Tree-Form Proof Representation:
  First, we presented the general concept and features of EUODHILOS systems. The proofs are represented in tree-form so that the users can easily capture what they are saying and how they are constructed.

- Proof Editing in Tree-Form:
  All the operations on the proofs are also performed in tree-form. The connection and separation are typical ones. To connect two proofs, we specify the conclusion part of a proof and one of the assumptions of another proof. Then the system automatically finds the proper substitution for the metavariables in these proofs and connects them if possible. To separate a proof fragment, we specify the node where the separation should occur. Then the fragment is divided into two smaller fragments. Separation is useful when a useful subproof structure is found in a big proof. After obtaining the useful part, it can be saved as a derived rule.

- Model of Proving Style:
  The most crucial difference of proof structures of EUODHILOS systems in comparison with those of other popular systems[27; 29; 85] is that the former are based on relational model and the latter on functional model. In the functional model, a rule is a function which gets a goal formula as its argument and produces a set of subgoals, in the backward deduction style. It does not give any guarantees to the validity of this reduction. A proof is finally verified when the proof tree is completed and the validation process from the leaf nodes to the root node finishes. In the relational model taken in EUODHILOS, a proof gives some relationship among its premises and conclusion. A rule is a primitive proof. Proofs may be connected by unifying the conclusion of a proof and a premise of another proof. In this connection process small proofs get together and grows to a larger and more complex proof. A proof finishes when all the premises are discharged.

  This difference is important for usability because more flexible proofs can be achieved in the relational model, and thus it is more appropriate for adapting to the user's reasoning style.

- Forward, Backward, and Mixed Reasoning:
  Our observations of human reasoning indicate that humans use both forward and backward reasoning mixed together. Forward reasoning starts with an assumption, axiom, or theorem in a proof window. These primitive proof fragments are used as premises and are combined in rule applications. The sizes of these proof fragments gradually increase as the reasoning proceeds.

  Backward reasoning starts with specifying a goal, which is a proof fragment consisting of a single formula that has not been proved to be a theorem. In backward reasoning, a rule application to a goal formula deduces one or more subgoals to be proved. Backward reasoning terminates when all the subgoals are proved to be axioms or theorems, or are to be discharged in rule applications.

  To support human reasoning, therefore, we must provide a mixed reasoning feature. Fortunately, mixed reasoning is fairly easy in our systems because the user can change the modes from forward to backward and vice versa at any time. This feature is a great advantage of EUODHILOS systems in comparison with other systems.

- Schematic Reasoning with Metavariables:
  Schematic proofs are also important in such considerations. Generally users cannot specify the goal formulas in detail in advance of the reasoning. EUODHILOS allows the users to use metavariables which are instantiated at any time in the reasoning. They are automatically instantiated during the manipulation of proof fragments such as rule applications and connection of proof fragments.

- Proof Reusability:
  Reusing proofs in the form of theorems/lemmas and derived rules contributes to make the reasoning easier to proceeds. EUODHILOS has two kinds of reusable data: theorems and derived rules. They are basically the same except theorems hold without any assumptions, whereas the derived rules have one or more assumptions. Once registered the theorems can be used like axioms and the derived rules like inference/rewriting rules. A facility for expanding the theorems and derived rules into original proofs is also available. Users can modify parts of the proofs and reuse them in other proofs.

  By using these as components of big proofs, the proofs become more compact so that users can recognize and construct such proofs easier.

Proof strategies of EUODHILOS can be applied to the proof fragments where we cannot predict how the proof proceeds precisely in advance. This is different from the proof strategies of other systems where users have to give sufficient parameters to perform the application. Further the tactics automatically find the axioms, theorems, and proofs in order to use them to the place where they are usable. This feature contributes to easy-to-use of tactics where users are not always able to specify all the information that should be used in the proof. These features indicate that it

is suitable to relatively small or medium sized straightforward proofs rather than proving big proofs. If such a tactic is applied to a big proof, the system may generate a huge number of candidates and it may be quite hard for the user to choose the most appropriate one. For reducing the burden of the user in a simple and boring proving operations such as repeating the similar processes and something like this, this style of semi-automated proving is considered to be a quite useful way of reasoning.

Chapter 7

Experiments: Application Examples

The aim of this chapter is to demonstrate the potential and usefulness of the EUODHILOS architecture. In order to achieve this aim we take up several logical systems and illustrate how the logic description framework of EUODHILOS can be effectively applicable to describing such a wide spectrum of formal systems and proof constructions[103] together with a lot of actual windows of the formulations and proof constructions. In spite of that most of the formal systems in this chapter are formulated as pure logical systems, EUODHILOS will also be applicable to wider formulation styles of formal systems, because essentially the wide variety of formal systems can and will be represented in terms of logical formulations. The formal systems and proof examples that we deal with in this chapter include various pure logical formulas, the unsolvability of the halting problem and an inductive proof with first-order logic, the proof of equivalence between the principle of mathematical induction and the principle of complete induction with second-order logic, modal reasoning about programs with propositional modal logic, the reflective proof of a metatheorem, Montague's semantics of natural language with intensional Logic, Martin-Löf's intuitionistic type theory, and reasoning about program properties with Hoare logic and dynamic logic. These logics constitute a currently well-known and wide range of logics or formal systems. See [100] for the detailed definition of each logic and more examples.