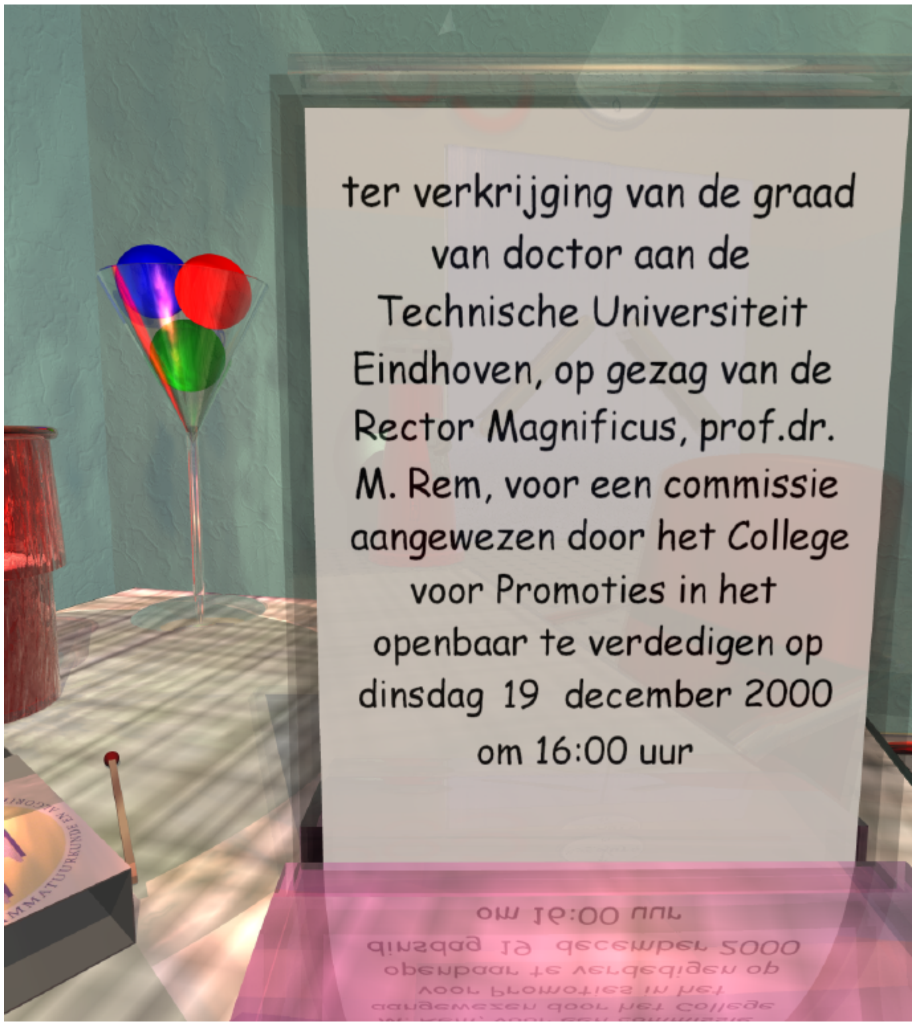
Cocktail : a tool for deriving correct programs

Cocktail
PTS
ATP
Hoare
A Tool for Deriving Correct Programs
SOBU
Samenwerkingsorgaan Brabantse Universiteiten
KUB
TU/e
M.FRANSSEN

Cocktail:
A Tool for Deriving Correct Programs

PROEFSCHRIFT



ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op dinsdag 19 december 2000 om 16:00 uur

door

Michael Gerardus Johannes Franssen
geboren te Heerlen

This thesis has been carried out under the auspices of the Institute for Programming Research and Algorithmics (IPA) and the Cooperation Center Tilburg and Eindhoven Universities (SOBU).

```
Franssen, Michael G.J.

Cocktail : a tool for deriving correct programs
/ by Michael G.J. Franssen. - Eindhoven
: Eindhoven University of Technology, 2000. Proefschrift.
- ISBN 90-386-0841-1
NUGI 856
Subject headings : Program proving / Lambda calculus /
                   Automated theorem proving/
                   Programming ; formal methods /
                   Formal logic
CR Subject classification (1998) : D.2.6, F.3.1, F.4.1,
                                   I.2.3, D.2.4
```

IPA dissertation series 2000-07
Cover design by M. Franssen
All images modeled and rendered with Bryce 3D^{tm}

Cocktail:
A Tool for Deriving Correct Programs

Michael Franssen

cocktail /'kɒkteɪl/ n **1** [C] an alcoholic drink consisting of a spirit, or several spirits, mixed with fruit juice, etc: *a cocktail bar/cabinet/lounge* ○ *a cocktail party.* **2** [C,U] a dish of small pieces of food, usu served cold at the beginning of a meal: *(a) prawn/shrimp cocktail* ○ *(a) fruit cocktail.* **3** [C] (*infml*)any mixture of substances: *a lethal cocktail of drugs.* See also `Molotov Cocktail`

(Oxford Advanced Learner's Dictionary)

Acknowledgements

First of all, I am indebted to Rob Nederpelt for teaching me how to write my documents in such a way that they are readable by people other than the author. His constructive comments and guidelines were truly invaluable during the writing of this thesis. However, I claim full responsibility for any unclarities and nonsense written in this book.

Also, I have to thank Kees Hemerik, who I believe was the true originator of this marvelous project, for offering me the Pascal code of the flexible bus system. I still owe him a favor for accepting that I translated the code into Java in an unattended moment.

I thank Harrie de Swart for introducing me to automated theorem proving, many stimulating discussions and numerous comments on early versions of this thesis.

I am grateful to Tijn Borghuis, who helped me to restructure several chapters of this thesis. I also thank Jos Baeten and Christine Paulin for being part of the reading committee and providing useful comments on the thesis. Furthermore, I thank all people of the it's-lunch-time-club, who make working at this university a lot of fun: Suzanna Andova, Roel Bloo, Dragan Bosnacki, Francien Dechesne, André Engels, Georgi Jojgov, Bart Knaack, Twan Laan, Susanne Loeber, Sjouke Mauw, Kees Middelburg, Marcella de Rooij, Martijn Oostdijk (TAFKATAM member), Anne-Meta Oversteegen, Peter Peters, Tim Willemse (TAFKATAM member) and Jan Zwanenburg.

Last but not least, I thank Linda Bothmer for putting up with my techno-talk about Cocktail for so long.

Michael Franssen

Contents

Chapter 1

Introduction

This thesis describes the design and implementation of Cocktail: A tool for Deriving Correct Programs. In Cocktail, programs are derived from their formal specification by stepwise refinement.

This simultaneous construction of a program and its correctness proof is based on the Dijkstra-Hoare calculus (see [Dij76]). The main advantage of this approach is that the programmer is guided to a correct program by the specification during the program's construction. The method has been developed and used for many years at Eindhoven University of Technology. It is an important instrument in the computer science curriculum: students learn to program using this method.

Up until now there is no tool support for the method. This means that all administration of proof obligations has to be done by hand, which is a lot of work for larger programs. Also, the proofs required to establish program correctness can be complicated. Since the correctness of the program depends on the correctness of the proofs, the latter have to be constructed very carefully. In short: the problem of program correctness is replaced by a problem of proof correctness. Therefore, we wanted a tool that keeps track of all proof obligations, supports the programmer during proof construction and checks the correctness of proofs and programs.

Such a tool has to fulfill many requirements, which have several origins. In order to keep a better overview of all requirements, we will discuss them from the perspectives of three representatives: the programmer, the logician and the system designer. The programmer, as the intended user of the tool, will be mainly interested in the usability. Hence, she[1] is concerned about the extent in which the tool allows her to focus on the program and about the extent in which the tool assists in constructing correctness proofs. The logician, as the theoretic guardian of the tool, is concerned about the formal basis of the tool and the correctness of its implementation. The system designer, who has to build the tool, will be concerned about its feasibility. It should be designed in such a way that it is usable, safe, maintainable and extendable.

Since the time for this PhD project is limited, we will not attempt to build an entire tool, but rather a proof-of-concept to show that it is actually possible to build a tool that meets our requirements. This proof-of-concept has sufficient functionality to support programming courses in the curriculum of Eindhoven University of Technology.

1.1 A Programmer's Tool

We want the tool to support the derivation of programs according to the Dijkstra/Hoare method. In this method, the program is derived from its specification, generating proof obligations on the fly. If we want the tool to be accepted by a programmer, we must avoid to prescribe an order in which the program is derived and the proofs are constructed. Also, a programmer wants to focus on the program, not on the correctness proof. The tool should support this. If the programmer has to prove a theorem, this should be as

[1]Whenever we need to use a personal or a possessive pronoun to refer to a previously mentioned person, we will use the female version. There are plenty of books were the male version is chosen.

easy as possible. Ideally, the tool should construct the proof automatically. If this is not possible, the proof must be constructed interactively in a way the programmer is familiar with.

To support this way of program derivation we need a Hoare logic combined with a theorem prover for first-order logic.

The Hoare logic is needed, since it is the basis of Dijkstra/Hoare program derivation. The Hoare logic links specifications, usually in first-order logic, directly to imperative programs. Since no intermediate encodings of programs are required, Hoare logic is relatively intuitive to a programmer.

The first order logic is needed to construct correctness proofs. If a programmer is familiar with a formal proof logic, it is usually this logic. Higher order logics, dynamic logics, modal logics etc. are more used by logicians and theoretical computing scientists. Also, if we want to support automatic proof construction, we need to keep the logic weak. Automated theorem proving for more powerful logics is almost impossible. First-order logic can be reasonably automated and is powerful enough to write meaningful specifications.

1.2 A Logician's Tool

A logician will be mainly concerned about the correctness proof of the correctness of the program. For this she will not blindly rely on the correctness of the tool used to construct this proof. But still, she requires that proofs constructed with the tool are guaranteed to be correct. Otherwise, using a tool will appear useless to a logician. The tool, however, will tend to become large, certainly if more and more bells and whistles are added during its construction. The logician will then lose the overview over the system and be unable to convince herself of the correctness of the entire tool.

De Bruijn was the first to tackle this problem, by proposing the Automath system. Proofs constructed with Automath are represented syntactically in a simple, uniform manner. This has the following advantages:

- The correctness of proofs can be verified after they are constructed. This is enabled by the simple representation of the proof, which actually follows the derivation rules of the formal system. The proof representation is chosen in such a way that the proof checker is small, simple and hence, reliable.

- Since proofs are represented explicitly by syntactic terms, they can be communicated to other systems. This allows the logician to build her own tool to verify the proof.

The requirement of communicable proofs is called the De Bruijn criterion. Examples of Modern Automath-like formalisms are Pure Type Systems, Edinburgh Logical Framework, and Martin-Löf Type-Theory. For each of these separate tools are available.

A drawback of this approach is that proofs have to be constructed in a very detailed way. When constructing a proof, the user usually wants to take larger steps than those allowed by the formal system. A tool can support this by offering larger steps and translating these into steps accepted by the used formalism.

Systems based on such formalisms are, almost without exception, interactive systems based on higher order logics. Automatic proof construction is barely supported. However, since automatic proof construction is important to the programmer, we want to combine an Automath-like systems with an automated theorem prover. On the other hand, this might violate the safety of the system. Therefore, we want to translate automatically generated proofs into the system's proof representation, just like we do with larger proof steps. This enables us to use automated theorem proving without extending the logic in an unforeseen way and moreover, we can verify the generated proofs afterwards.

Besides the proof logic, the tool also has to support a programming logic. Usually, those logics do not conform to the De Bruijn criterion. A possible solution for this is to encode the entire Hoare logic within the theorem prover logic. This would have the following disadvantages:

- The programmer has to encode all programs in the theorem prover's logic. This shifts the focus from program towards proof, which the programmer does not want.
- Embedding the Hoare logic usually requires higher-order logic. Supporting automated theorem proving will then be harder.
- It takes more efforts to build a tool for programs embedded in a logic than a tool that can manipulate programs directly. Given the time constraints of this project, this is undesirable.

We therefore want to keep the Hoare logic separated from the proof logic and combine them by using additional derivation rules. Preferably, we want to develop a Hoare logic that can be used to verify complete programs and that conforms to the De Bruijn criterion.

1.3 A Designer's Tool

Building a fully featured tool is not possible in the time available in a PhD project. We therefore want to build a proof-of-concept of a tool. However, we do want the tool to be powerful enough for educational purposes. Using the tool in programming courses will also yield feedback from students, which will help to guide future development of the tool. For these reasons, we choose to build support for:

- A simple While-language instead of a full Pascal-like language. This will suffice for educational purposes.
- First-order predicate logic represented by a Pure Type System (PTS) as the specification language. This logic is well known amongst programmers and the PTS formalism allows future extensions. Also, in PTSs, proofs can be verified through type checking and the system will conform to the De Bruijn criterion.
- A simple tableau-based automated theorem prover. However, we then need to be able to convert closed tableaux into λ-terms of the PTS to allow them to be checked.

Since initially we are only building a prototype, we want the tool to be extendable, such that in the near future a full imperative programming language and in the farther future perhaps an object-oriented language can be supported.

To keep the tool extendable, we have to implement it in a transparent manner. Therefore, we choose to design the tool modularly, such that parts of it can be studied, maintained and replaced independently of each other. Also, this supports the safety of the tool, since the type-checker will be a separate, hopefully small, part of the tool that is carefully implemented independently of the rest of the system.

We choose to implement the tool in an object-oriented language for the following reasons:

- Modularity is better supported in an object-oriented language than in any other kind of current programming language.
- By using inheritance, the tool can be smaller and therefore, hopefully, more reliable.
- Object oriented languages offer better support for building graphical user interfaces than functional or logical programming languages.

We have chosen to implement the tool in Java, since this language is platform independent. Java also comes with extensive standardized libraries for building graphical user interfaces.

Globally, the tool consists of three parts:

A Symbolic Engine: This is the heart of the system. It consists of the term representations and type checkers of the formal systems. Everything proved by the user has to be represented in the symbolic engine and is checked for correctness there. The symbolic engine ensures safety and conforms to the De Bruijn criterion.

A Tactic System: To make proof construction easier, the tool offers support for larger proof steps than allowed by the symbolic engine. Each of these larger proof steps is automatically translated into a term of the symbolic engine to ensure safety of the system. The tactic system also contains all means for automatic proof construction. It dictates how the tool can be used and hence, it should support several styles of reasoning.

A User Interface: To enable a user to work with the tool, a user interface is needed. The cocktail of formalisms in the tool makes interaction relatively complicated. Changes in the program result in changes in the proof-obligations and have to be visible immediately. Therefore, the user interface has to display all available information consistently. Also, all displayed information has to be correct at any moment. We want to realize this through a number of connected windows that display all information in a suitable way, i.e. programs are displayed differently from proofs. Changes in one window automatically lead to changes in other windows. Interaction should take place with the displayed information directly instead of indirectly through commands, issued from a separate window.

1.4 Overview of this Thesis

The thesis is divided in three parts:

Part I: This part discusses the requirements of the different representatives and the way in which we want to meet these requirements. The programmer's requirements, the logician's requirements and the system's designer's requirements are separately discussed in chapters 2, 3 and 4 respectively. We also compare our requirements with the state of the art in literature and motivate our design choices.

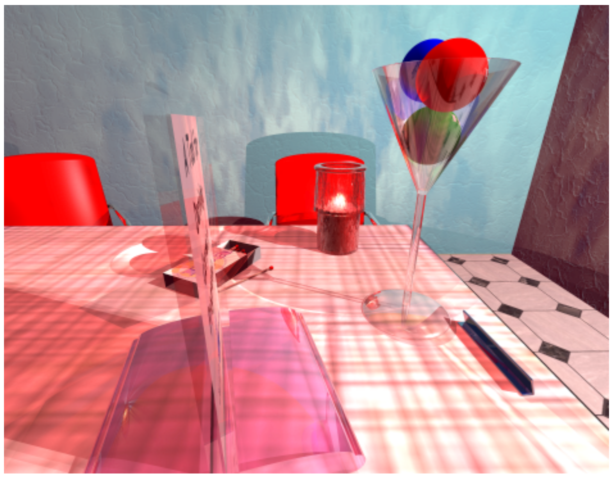
Part II: In this part all required formal systems are discussed. Since all formalisms used by the tool are based on first-order logic, the definition and semantics of this logic are given in chapter 5. Chapter 6 describes our design of an accurate model of first-order logic in a PTS. Methods for automated theorem proving are discussed in chapter 7. The programming logic used by the tool is defined in chapter 8. Since the formalisms discussed cannot be used directly in the way envisioned for our tool, we show how to combine the formalisms in chapters 9 and 10: Chapter 9 describes the translation of closed tableaux into λ-terms and chapter 10 links the Hoare logic to a PTS.

Part III: Once the requirements of part I are known and the formal basis of part II is available, we can design the actual tool. Chapter 12 discusses some design considerations and resolves two final issues in the theory required for the tool. Chapter 13 states the design goals more elaborately and extracts a number of operational requirements from those goals. In chapter 14 the design of the graphical user interface is discussed. It deals with formula-, proof- and program-display, as well as interaction issues. In chapter 15 the actual implementation is briefly discussed, casting some light on the modular design at several levels. Finally, the results are discussed in chapter 16.

The thesis does not need to be read entirely by everyone interested in Cocktail. People interrested in the main features of the tool and the motivations behind them should read part I. For readers who are mainly interested in the theoretical background, reading part II will suffice: people interested in combining interactive and automatic proof construction should start with chapters 7 and 9. Those more interested in the programming logic and its connection to a theorem prover should read chapters 8 and 10.

Part I

Perspectives to Deriving Correct Programs



Chapter 2

A Programmer's Perspective

Our aim is to build a tool that assists programmers in constructing correct programs. However, to decide when a program is correct, the programmer must provide a formal specification of what the program is supposed to do. Ideally, the tool should automatically derive a correct program from this specification, but this is only possible for the simplest programs. For instance, H. Christensen describes a tool for automatic program derivation in [Chr93], but this tool has only a limited capacity to use quantifications in specifications. As a result, fibonacci numbers or factorials cannot be defined. For such functions, at least first-order logic is needed and since this logic is undecidable, we need an interactive tool to be able to handle meaningful programs.

The programmer will be interested mainly in the program itself and less in its correctness proof. To accomodate the programmer, the tool needs to represent programs as directly as possible and has to construct proofs automatically whenever possible.

In this chapter, we will discuss which methods are available to the programmer to prove correctness of her programs and which requirements this imposes on a tool.

2.1 Approaches to Correct Programming

Which methods are available to the programmer to prove correctness of her programs? In literature, several methods have been proposed. We distinguish them here by the order in which the program and its correctness proof are constructed.

2.1.1 Constructing the Program first

In this approach, the correctness of the program is verified after it has been written. There are several ways to do this:

Model checking: A tool simulates all possible executions of the program and checks if all established states satisfy the specification. If a case is found in which the program produces faulty results, the trace leading to this result is reported to the user. The program and its specification have to be formulated in the tools language for this purpose. Although these languages may resemble regular programming languages, they are typically much more limited. Hence, the programmer still has to code her program in an artificial way.

Theorem proving: In this method, the program is first formalized within a theorem prover and then its correctness is proved interactively. However, this forces the programmer to encode the program in a theorem prover logic and hence, this method is not intuitive to the programmer.

Verification Condition Generating: In this method, proof obligations are automatically computed for a formally annotated program. These proof obligations have to be satisfied afterwards. To compute the proof obligations, a Hoare logic is used and consequently, the focus is mainly on the program.

A general drawback of constructing the program first is that no support is available during the program construction process. Errors are only detected after the program is completed.

2.1.2 Constructing the Proof first

In this method, the programmer first has to formally prove a theorem, which indirectly states that there exists a program that meets the specification. Among other restrictions, the proof must be constructive in a certain way. A tool then automatically extracts a program from the proof of this theorem. How this is done exactly is beyond the scope of this thesis, but the interested reader is referred to [PM89]. This method is hardly suitable for a programmer, since it focuses entirely on the proof, not on the program. The extracted program can be very different from the programmer's expectations and wishes. For instance: it might turn out to be very inefficient. Since the program is not visible during the construction of its proof, the only way to influence the outcome is to guide the construction of the proof. However, this requires the user to be an expert in program extraction and theorem proving. Such users, however, are logicians, not programmers.

2.1.3 Simultaneous Construction of Program and Proof

In this method, the program and its correctness proof are constructed hand in hand. It is based on a Hoare logic, which directly links the program to its semantics. From Dijkstra-Hoare calculus and the works of Gries, Feijen and Kaldewaij [Gri81, DF88, Kal90] it becomes clear that programs can actually be *derived* from their specifications. When finished, the program and its correctness proof are both available.

The initial specification is rewritten over and over again, inspiring new pieces of program to be inserted along the way, yielding sub-specifications. This process is sometimes referred to as (program) *refinement*. The Hoare logic enables the programmer to construct proofs in any order and at any time during the program derivation. The program under construction is available the whole time. Poll developed a similar calculus for functional languages in [Pol94], but for this calculus there is less practical experience with deriving programs than is available for Hoare logic. Also, we want to support imperative languages rather than functional languages.

2.2 Existing Tools

For all methods presented in the previous section, some tools are available. We will discuss them in the same order.

2.2.1 Constructing the Program first

We have several options:

Model checkers: Model checkers, like SPIN [Hol97], have been applied successfully to verify for instance communication protocols. However, model checkers suffer from their non-scalability: since all traces of a program are considered, the number of possible states grows exponentially. This is known as the state-explosion problem. Since the method is also not suitable for interaction, it will not be possible to apply it to more complex algorithms. Also, programs have to be written in the tool's language, in SPIN's case Promela, which resembles, but is not equal to, a regular programming language.

Theorem provers: COQ [Coq97], LEGO [LEG97], PVS [ORS92], Yarrow [Zwa98, Zwa97] and other theorem provers are suitable to construct correctness proofs. However, this invariably requires the program to be encoded within the theorem prover (using a deep or shallow embedding as set forth in chapter 10 of this thesis). This is not interesting to a programmer, to whom a Hoare logic is just about acceptable. Also, since suitable theorem provers are usually based on higher order logic, automated theorem proving is barely supported. All proofs have to be constructed manually with the tool.

Verification Condition Generators: A tool like Sunrise [GGH98] allows the programmer to prove correctness of a program without her having to encode it first. It takes the program, annotated with invariants, as input and generates a set of verification conditions. These have to be proved correct within a theorem prover; which, in case of Sunrise, is the HOL system. The generated conditions are typically very complicated, since the VCG combines many separate proof obligations into a few theorems. To the programmer, the relation between the program and the verification conditions will not alway be clear. If the original program or specification was not correct, the verification conditions barely provide any hints for improvements. The advantage of verification condition generators is that they are able to deal with complex programming constructs, like mutually recursive procedures (see [HM96]).

As stated before, a general drawback of constructing the program first is that no support is available during the programming process.

2.2.2 Constructing the Proof first

The only tool for this method that we know of is COQ. The partial program is not visible during the construction of the proof and can be disappointing afterwards (it can be an unintended solution, turn out to be inefficient etc). Also, the programmer has to use a theorem prover's interface, which she is not used to. Moreover, the result of program extraction is always a functional program; in case of COQ, a program in ML. However, we want to support imperative programs.

2.2.3 Simultaneous Construction of Program and Proof

There are only a few tools. We will discuss two of them:

The Karlsruhe Interactive Verifier: This is a large system, which is based on dynamic logic. The interface is designed for proving logical theorems. Even though the Karlsruhe Interactive Verifier (KIV) is mainly a *verification* system, M. Heisel also implemented Gries' method in it to *derive* programs (see [Hei92]). However, since this is not an integrated part of the system, the presentation of the program derivation process is not suitable for the programmer. Moreover, only few programmers know how to use dynamic logic.

The Refinement Calculator: This tool is implemented as an extension to the HOL theorem prover; a theorem prover for higher order logic (see [GM93]). Because of its logic, it barely supports automated theorem proving. Also, rewriting is supported only in a limited way, even though rewriting is used much in Dijkstra-Hoare calculus. Moreover, the programmer is confronted with a theorem prover's interface. Programs are encoded in HOL's logic using a shallow embedding as described in chapter 10 and hence, are less readable to the programmer.

The language supported by the refinement calculator is not defined explicitly. For instance, it is possible to use functions within a program that were intended for specification purposes only. These functions might even be non-computable. As a result, it is not always clear if and when the development of proof and program is complete.

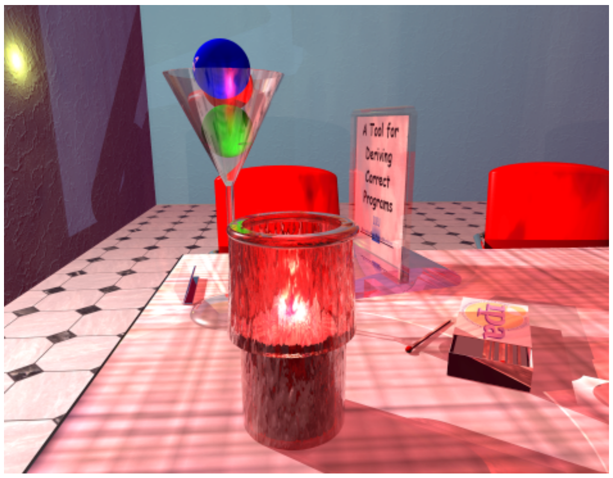
2.3 Required Ingredients

The discussion above shows that, according to the programmer, the tool needs at least the following two components:

A Hoare Logic, which is needed to write programs. This logic is widely known amongst and accepted by programmers, as opposed to dynamic logic. Also, many meta-theoretical properties of Hoare logic are known and many advanced programming constructs can be supported (For instance, mutually recursive procedures as described in [HM96]).

A Theorem Prover, which is needed to solve the proof obligations generated by the Hoare logic. Preferably, the proofs should be constructed automatically. The theorem prover should also provide extensive support to rewrite specifications.

A tool with these characteristics does, to our knowledge, not yet exist.



Chapter 3

A Logician's Perspective

This chapter explores the wishes of the logician with respect to a programming tool. In the first section, a logic is chosen, based on the requirements of the programmer. Then, additional requirements are formulated for the logician. The second section discusses why existing tools do not satisfy our requirements. The last section is a brief list of the requirements gathered for the tool so far.

3.1 Choosing a Logic

In chapter 2, we found that to write useful specifications, we need at least a first order logic. Since validity of first order formulas is undecidable, we have to be able to support interactive proof construction.

The programmer wants to use automated theorem proving whenever possible. Hence, we should avoid using a logic stronger than first order logic. Also, a weak logic has more meta-theoretical properties that can be utilized by a tool and has semantics that are easier to grasp for non-logicians.

In chapter 2 we argued the need to support a Hoare logic. The requirements of the programmer also imply that we should avoid to embed this Hoare logic in the theorem prover's logic. Embedding the Hoare logic would probably force us to use higher order logic. Also, we would force the programmer to encode her programs in the proof logic, which again shifts the focus to theorem proving.

Whatever formal system is supported by the tool, the logician wants a guarantee of correctness. Errors in the tool should never cause errors in final proofs. For instance, the Boyer-Moore theorem prover (see [BM88]) uses many advanced techniques to automatically find a proof of a theorem, but only reports if the theorem was correct, incorrect or if the tool could not solve the problem. There is no way to verify the correctness of the proof. PVS, an interactive theorem prover, suffers from the same problem. In PVS it is not exactly known what logic is supported, since several parts of the system use different rules (repeatedly contradictions were proved within PVS, always leading to improvements of the tool).

The first person to tackle the safety problem of implemented logics was De Bruijn, who proposed Automath (see [NGdV94]). In this system, the proofs are represented syntactically in a simple, uniform way that follows the structure of the logical rules. The advantages of this approach are:

- Proofs can be easily verified once they are complete, regardless of the way they were constructed. One only has to check the correctness of the syntactical representation of the proof.

- The proof can be communicated to other systems. If a logician does not believe the original system is correct, she can write her own proof checker, which typically is small, simple and, if needed, can be formally proved to be correct. This is called the De Bruijn criterion.

A drawback of this approach is that proofs have to be constructed in a very detailed way, which makes them long and, certainly to the programmer, unattractive. Modern versions of Automath-like systems are Pure Type Systems (PTSs, see [Bar92, Ter89]) implemented in, for instance, Yarrow (see [Zwa98]), Edinburgh Logical Framework (see [HMP93]) implemented in ELF (see [Pfe91]) and Martin Löf type theory (see [ML82]), implemented in Nuprl (see [CAB86]).

Because the level of detail in formal logics is too high, the system must provide support for larger proof steps. To ensure safety of the system, these larger steps have to be broken down into small steps accepted by the formal logic. A good example is an automatically constructed proof. If we can encode generated proofs in the formal logic, we can safely combine automated and interactive theorem proving.

Hence, we want the tool to support first-order logic, both interactive and automated theorem proving and it must conform to the De Bruijn criterion.

3.2 Existing Tools

Basically, there are two kinds of tools. They can be distinguished by the logic they support:

First Order logic: Theorem provers for first order logic are usually based on either the resolution or the tableau method. Examples are JAPE [BS], Otter [McC94], 3Tap [HBG94] and Bliksem [dN95]. A drawback of these tools is that they are not interactive and hence, incomplete. An exception is JAPE, which is interactive. However, JAPE, like all other mentioned systems, does not conform to the De Bruijn criterion.

Higher Order Logic: Since automated proof search is barely possible, these tools are all interactive. Examples are COQ [Hue89], Yarrow [Zwa98], LEGO [LEG97], Nuprl [CAB86], TypeLab [HLS97], HOL [GM93], and PVS [ORS92]. We compare each system against our requirements of safety, support for rewriting and automated theorem proving.

Safety: PVS does not provide representations of the actual proof and hence, does not provide the required safety. Type checking in PVS differs from type checking in λ-calculi and is undecidable: type checking a term in PVS can result in proof obligations, which have to be solved within PVS itself. Hence, PVS's safety check depends on PVS's safety. COQ, Yarrow, LEGO, Nuprl and

TypeLab all provide proof representations and use type-checking to ensure safety.

Rewriting: Rewriting in COQ, Yarrow, LEGO, Nuprl, TypeLab and HOL is only supported in a limited way. Once Leibniz equalities are formulated in higher-order encodings, they can only be used to rewrite the current goal of the theorem under construction. If several applications of the same equality are possible, determining the desired rewriting is complicated.

Automated Theorem Proving: Automated Theorem Proving (in short ATP) in these systems is hardly possible, since they are based on higher order logic. PVS does provide many facilities for ATP, but is not suitable for our purposes by lack of safety. In [BHN00] a method to translate proofs of the automated theorem prover Bliksem into proofs of the interactive system COQ is proposed. Unfortunately, translation is not always possible, despite COQ's powerful logic. Axioms are needed to deal with skolemized functions of the resolution proof. Accepting Bliksem's proofs within COQ then violates the safety and the De Bruijn criterion.

3.3 Required Ingredients

The logician requires the tool to support the following:

Safety: The logic's implementation should be safe and satisfy De Bruijn's criterion. Hence, we need to find a type-theory for the first-order logic we have chosen to use.

Interaction: Since theorem proving in first-order logic is undecidable, the user must be able to interactively construct proofs. Also, rewriting must be supported extensively.

Automated Theorem Proving: The programmer wants the tool to construct proofs automatically whenever possible. The logician demands safety of the system. Hence, automatically constructed proofs have to be translated into a proof representation of the formal system used by the tool.

A tool with these characteristics does, to our knowledge, not yet exist.

Chapter 4

The System Designer's Perspective

The programmer and the logician outlined the requirements of the formal foundations of the tool. When implementing a tool, many design considerations enter the stage. These are usually more practically oriented than the requirements given until now. Therefore, this chapter discusses the system designer's perspective, whose task is to construct a tool that does not only meet all requirements of the programmer and the logician, but which is also usable to the students for which it is built initially. Since in the future, many more people should be able to use the tool, the system designer has to consider not only the initial requirements, but also how future development of the tool should take place. As a result, decisions have to be made about the basic characteristics of the tool, the implementation language, the global design and the user interface. Once these decisions are made, we can give the full list of requirements of the tool.

4.1 General Considerations

Since the time for this PhD project is limited, we will have to restrict ourselves to building a tool as proof of concept. Implementing support for an entire programming language (like Pascal) would not be possible in four years, since we will also have to develop some theory. Also, it will not be necessary: a simple *While* language will suffice to support first and second year programming courses at Eindhoven University of Technology. Using the tool during these courses will also provide feedback for future development of the tool. To enable such development, the tool needs to be extendable. For the near future our goal is to implement the full GCL, to be able to use the tool to support more advanced programming courses. In the long run, the tool might even evolve into a real-world programming tool.

4.1.1 Generic versus Adjustable

In practice, two methods to build extendable tools for formal systems can be distinguished. We will briefly discuss their advantages and drawbacks.

Generic tools: Generic tools take a description of the formal system as input and generate support for this system (for instance: a description of a Gentzen-style logic). The most important advantage of this approach is that the formal system supported by the tool can easily be altered by a user: one only has to change the system's input. The main drawback is that the tool cannot utilize any meta-theoretical properties of the formal system. After all, it is not known beforehand which formalism will be used. Therefore, a generic tool is hard to build: one has to provide meaningful support for an unknown formal system. Examples of generic tools are JAPE (see [BS]) for Gentzen-style logics and ASF+SDF (see [vdBvDK96]) for algebraic systems.

Adjustable tools: Adjustable tools do have a formal system built in, but are constructed in such a way that these can be changed and extended relatively easy. However, changing the formalisms supported by the tool will be harder than for generic tools, since it will require altering of the tool. On the other hand, it is still possible to provide switches to enable or disable certain parts of the formal system. In that case, the formal system is not restricted to a single fixed logic, but at least all possible systems that can be supported by the tool are known during its construction. This enables the system designer to exploit meta-theoretical properties of the formal system(s). Hence, it will be easier to provide meaningful support in these systems. A nice example of an adjustable system is Yarrow, which provides support for all single sorted Pure Type Systems (see [Zwa98]).

We choose to build an adjustable system for the following reasons:

- In practice, most generic tools are only used for a single logic. Usually, when designing a generic system, it requires so much time to build support for the initial logic under consideration, that there is hardly time to to consider entirely different logics. However, ASF+SDF has been applied successfully for many different systems. Unfortunately for us, ASF+SDF is aimed at algebraic systems, not at logics.

- The prototype must be built within the given amount of time for a PhD project. It is important to have a working proof of concept, so we cannot take the risk to end up with a generic tool that is not yet powerful enough to be used in an educational setting. Since it is easier to support a single logic, building an adjustable tool will yield results faster. For instance, it took about ten times as long to build the ASF+SDF system than the time available for this project. If we want a working proof of concept, a generic tool is just not feasible.

In the exceptional case where a user wants to change the logic, she has to learn about the implementation of the tool. However, this implies that the tool must be designed carefully to remain transparent. This, in turn, raises questions about how the tool should be implemented.

4.1.2 Choosing a Paradigm

Before we start building the tool, we have to choose what language we will use to built it. In this subsection, we will select the software construction paradigm (or programming paradigm). In the next subsection, we will choose the actual implementation language.

To allow future extensions of the tool it is necessary

- for the tool to be comprehensible (transparent)
- that parts of the system are independent and replaceable (modular). For instance, the type-checker should be a small separate module.
- to be able to re-use parts of the tool that are already implemented.

Hence, it is recommended to design the tool in a modular way, such that parts of the tool can be studied, maintained and replaced independently from other parts. It is also necessary to support re-use of the code.

There are two paradigms that are obvious candidates for the construction of our tool. The following table summarizes their most important properties related to our wishes stated above.

Functional Programming	Object Oriented Programming
• similar to λ-calculus	• requires more work to implement new data-types
• re-use of code is non-trivial	• re-use of code is supported through inheritance
• user interfaces are still rather difficult to build	• user interfaces can easyly be build (certainly when using visual tools)

Since modularity and re-use of code are essential in our tool, we choose to use an object oriented programming language. This also allows us to build a good user interface more easily.

4.1.3 Choosing a Language

In our choice of the actual implementation language, we consider the following issues:

- We want the tool to be platform-independent for the following reasons:
  - The potential users (of the initial version and all future versions) of the tool use different platforms and operating systems.

  – We want the initial tool to work on smaller stand-alone machines (like student's computers) themselves, not on a network or mainframe. Future versions of the tool, however, may require more resources, forcing a move to a different platform.

- Since interaction with a tool like this will be non-trivial, we want extended features for building a user interface.
- The language should be clean and readable.

Object oriented languages suitable for platform independent programs are C++ and Java.

C++ programs run much faster than Java programs, but considering the interactive character of our tool this is not very important. The computer will have plenty of time for computations between the user's actions, especially for the relatively small computations it has to do. Moreover, C++ is not as standardized as it seems, especially when using its modern features like templates and exceptions. Also, C++'s libraries for building (graphical) user interfaces differ between platforms.

On the other hand, the design of Java is much more consistent. Also it provides extensive platform-independent libraries to build graphical user interfaces. Java is also freely available for many different platforms. Hence, we will implement the system in Java.

4.2 Main Modules of the System

The safety of the tool, required by the logician, leads to very detailed proofs. Since the programmer typically wants to use larger steps, we have to translate these into several smaller ones. A system designer wants to do this transparently and hence, in a modular way. Also, we should realize that the translations are independent of the interaction with the system. Therefore, we choose to design the system roughly in three parts, each extending the previous one:

The symbolic engine, which ensures the safety of the system.

The tactic system, which translates larger proof steps into smaller ones that are accepted by the symbolic engine.

The user interface, which enables the user to interact with the system by sending commands to the tactic system.

Each module is presented in a separate subsection.

4.2.1 Symbolic Engine

The symbolic engine consists of the representation of terms of the formal systems and the type-checkers. The type-checkers have to verify whether or not a term can be derived in the formal system to which it belongs.

The symbolic engine ensures that the entire system conforms to the De Bruijn criterion. Hence, it is very important to keep this module small and correct.

For first-order logic we will use a simple Pure Type System (PTS) as described in chapter 6. To safely support Hoare logic, we design a specific version in chapter 10 that conforms to the De Bruijn criterion. Also, we design this Hoare logic in such a way that it has the same structure and properties as a PTS. This allows us to re-use code of the PTS type-checker for the program checker. Hence, the symbolic engine remains small, comprehensible and can therefore be trusted to be correct. Also, having both systems represented by a uniform structure is more satisfying aesthetically.

4.2.2 Tactic System

The tactic system is a layer on top of the symbolic engine, which enables the user to use larger steps in the proof than those allowed by the formal system. All these larger steps are immediately translated into small steps that are accepted by the symbolic engine. Constructing proofs directly with the symbolic engine would be possible, but results in proofs that have too much detail. To the users this is too cumbersome and this causes them to lose the overview of their proof.

The actions a user can perform with the tool depend on the translations the tactic system is able to make. Hence, the tactic system dictates how the tool has to be used. Since we do not want to force one specific method of proof construction upon the user, we want the tactic system to support at least the following:

Backward Reasoning In backward reasoning, the goal of the proof is decomposed into new goals by applying known theorems to it. For instance, if one has to prove B, one can apply an already known theorem $A \Rightarrow B$ to the goal after which the new goal A remains to be proved. This is the usual method of proof construction supported by systems based on type-theory.

Forward Reasoning In forward reasoning, one derives new information from information that was already available, independent of the current goal of the proof. For instance, given $A \Rightarrow B$ and A, we may want to conclude B, even if this is not the goal of our proof. Since, to the user, there is no apparent reason why this should not be possible, she will expect it to be supported.

Automated Theorem Proving (ATP): We need this to alleviate the user from constructing many (nearly) trivial proofs. Note that since automatically constructed proofs are translated into terms of the symbolic engine as described in chapter 9, the automated theorem prover is part of the tactic system. Also, we want that ATP can be invoked by the user at any given moment to prove a (partial) theorem.

Equational Reasoning Since equational reasoning is often needed when using Dijkstra/Hoare calculus, we have to support it in our tool.

4.2.3 User Interface

The user interface lets the user communicate with the system. With the user interface, the user sends commands to the tactic system to construct proofs. Also, it shows all information about the program and its correctness proof available at any given moment.

The cocktail of formalisms supported by the tool is not a trivial one. Any changes in the program cause changes in the set of proof-obligations. Interactions with a system like this therefore have to be well thought through and have to be presented orderly to the user. Hence, it is necessary to visualize information in a comprehensible manner. Also, the information must be consistent and up-to-date at any given moment. The programmer will expect programs to be displayed different than proofs. To tighten the gap between user's expectations and the actual system, we want to use notations that the user is accustomed to.

Therefore, we want to build a graphical user interface (GUI), which uses several windows to display different pieces of information in a comprehensible way. The information in the windows is interconnected, such that consequences of an action are immediately visible everywhere. Interaction with the system should take place with the displayed information as directly as possible and not, as is often the case, through commands issued from a separate window. For instance, the user should be able to graphically select a part of a formula that she wants to rewrite in equational reasoning.

4.3 Required Ingredients

In short, the requirements of the tool are the following:

- The tool should be a proof of concept of a tool for deriving correct programs. Hence, we will restrict ourselves to using first-order logic and a simple *While* language.
- The tool must be adjustable. That is, transparent, maintainable and extendible, but not generic. We will establish this by creating a modular design and re-usable implementation of the tool.
- The tool should be platform-independent and be able to run on an average sized computer. Platform independence is established automatically through the use of Java as our implementation language.
- The tool must have a safe formal foundation that is implemented in a trustworthy way. Safety is obtained by using a typed lambda calculus, which conforms to the De Bruijn criterion and allows verification of the actual proofs by type checking. The required type checker is a small and simple program, which can be formally verified if necessary.
- Interactive theorem proving (both forward and backward) as well as automated theorem proving must be supported. Also, the tool should have extensive support for rewriting.
- To be useful, the tool must have an intuitive, comprehensible user interface that allows direct interaction with the displayed information.

A tool with these characteristics does, to our knowledge, not yet exist.

Part II

Theoretic Background

Chapter 5

Introduction

5.1 Purpose of Part II

The purpose of this part is to provide a theoretical foundation for a tool for the derivation of correct programs. This foundation consists of three parts: Firstly, an interactive theorem prover for first order logic based on type theory. Secondly, an automated theorem prover, embedded in the interactive theorem prover. Thirdly, a Hoare logic, which is linked to the theorem prover. The theorem prover will be used in the programming environment, but only to perform correctness proofs, not to model the semantics of the programming language. The derivation of programs is supported by a Hoare logic and is presented in chapter 8.

An interactive theorem prover with a high degree of automation combines the fields of interactive theorem proving and automated theorem proving. Both approaches to theorem proving have advantages and drawbacks. We will briefly present them here.

The reason for comparing the two is that we want to create a proof system that combines the advantages of interactive theorem proving with the advantages of automated theorem proving. If possible, we want to have the best of both worlds.

5.1.1 Interactive Theorem Proving

In interactive theorem proving one often uses typed lambda calculi, since these calculi provide an easy way to represent unfinished proofs as lambda terms with typed holes. The holes then have to be filled with terms of the correct type to complete the proof. Besides, typed lambda calculi have the following advantages:

- Verification of proofs is possible by type checking.
- Proofs can be communicated as λ-terms, since λ-terms are a standard representation of a proof.
- There is a uniform treatment of first order logics and higher order logics.

Examples of interactive theorem provers based on typed λ-calculi are Nuprl (see [CAB86]), COQ (see [Hue89]) and Yarrow (see [Zwa98]). For a more elaborate overview we refer to [Fra97].

A drawback of interactive theorem provers based on typed λ-calculi is that automating the proving process is hard. The reasons for this are, among others,

- Typed λ-calculi are usually used to model higher order logics. For automatic proof search in higher order logics no efficient methods have been found yet. Also, in higher order logics, even simple concepts like logical "and" and logical "or" are represented by complex higher order formulas. Therefore, even proving simple theorems is difficult from the computer's point of view.

- Many automated theorem proving methods use semantical concepts during proof construction. In typed λ-calculi, every proof has to be explicitly given as a λ-term. Hence, taking shortcuts in a proof, based on meta-theoretical properties of the logic, is not possible.

- Automated theorem provers usually do not provide a representation of the actual proof. That is, they only answer the question whether or not the theorem is correct. They do not show how this answer was computed. In λ-calculus the proof term always has to be stated explicitly and can become very large.

5.1.2 Automated Theorem Provers

Automated theorem provers (ATPs) are usually based on the method of tableaux or on resolution methods (see [BS98]), since these methods are powerful enough to attack problems formulated in first order logic. Besides, it is known that for every valid formula in first order logic both a tableau proof and a resolution proof exist, although it may not always be computable. Both methods are based on classical first order logic[1].

Automated theorem provers are not the answer to all our problems. In the following we will give a brief list of the most important drawbacks of ATPs:

- There are limits to what an ATP can prove. They can prove nontrivial theorems, but not (yet) hard theorems. They are most beneficial in proofs that are not hard but tedious (e.g. a proof consisting of many simple case distinctions in which the ATP can prove the separate cases). The Boyer-Moore theorem prover (see [BM88]) has been used to proof many theorems, but for this the theorems had to be manually split into numerous smaller theorems that could be handled by the tool. Therefore, the Boyer-Moore theorem prover is considered to be a tool to check proofs rather than a tool to construct proofs.

- Proofs constructed by an ATP are usually not suitable for a human reader. This gives rise to problems when the user has to interact with the ATP if the ATP cannot find a proof fully automatically. The user will then hardly be able to see where the difficulties arise and how

[1]There are versions of both methods that are suitable for intuitionistic and modal first order logic, but these are not standard. We will not consider these methods in this thesis, but the interested reader is referred to [dS93] and to [DGHP99].

they should be solved. Once the user has interacted with the ATP, the proof is no longer automatically reproducible. If the ATP had to reproduce the proof, it would again not find it fully automatically. To find the same proof again, the same user interaction as before is required.

- Since the representation of the actual proof is not standard, it cannot be communicated to other proof systems. These systems can therefore not be used to verify the automatically constructed proof. Tableau based theorem provers do construct a tableau as a representation of the proof, but this is not a communicatable representation. Resolution methods do not produce a representation of the proof at all, although many implementations provide an ad hoc (non-standard) internal representation for use within the same system.

A user benefits from an ATP the most if it is embedded in an interactive system which assists in proving hard theorems. The user can then invoke the ATP to deal with tedious or simple parts of the larger proof. This is exactly what is made possible with the system $\lambda P-$ introduced in this part: combining meaningful ATP with an interactive theorem prover based on a typed λ-calculus.

5.2 First Order Predicate Logic

Since in this part we deal with first order predicate logic, we will formally introduce first order logic. All formal systems used by the tool will have semantics formulated in this logic. Hence, it connects the formal systems at the meta-level. Therefore, it is important to describe this logic accurately.

In literature, formulas of first order predicate logic are defined as follows:

Definition 1 (First order formulas)
Let $\mathcal{F}$ be a set of function symbols, each with a fixed arity ≥ 0. Furthermore, let $\mathcal{P}$ be a set of predicate symbols, each with a fixed arity ≥ 0. For convenience, we assume a special predicate symbol Falsum *to exist in $\mathcal{P}$ with arity 0. Finally, we assume the existence of an infinite set V of variables. The sets $\mathcal{F}$, $\mathcal{P}$ and V are disjoint.*

Then the set T of terms is defined recursively as:

1. *$V \subseteq T$*
2. *If $f \in \mathcal{F}$ with arity n and $t_1, \ldots, t_n \in T$, then $(ft_1 \ldots t_n) \in T$.*

The set Prop of formulas is defined recursively as:

1. *If $P \in \mathcal{P}$ with arity n and $t_1, \ldots, t_n \in T$, then $(Pt_1 \ldots t_n) \in Prop$.*
2. *If $P, Q \in Prop$, then $(P \wedge Q) \in Prop$, $(P \vee Q) \in Prop$ and $(P \Rightarrow Q) \in Prop$.*
3. *If $P \in Prop$, then $(\neg P) \in Prop$.*
4. *If $P \in Prop$ and $x \in V$, then $(\forall x.P) \in Prop$ and $(\exists x.P) \in Prop$.*

We use the variable convention, which means that:

1. *Bound variables will always have a name different from free variables.*
2. *If formulas differ only in the names of their bound variables, they are considered to be equal (α-equality).*

Note that the special symbol *Falsum* will be important, even though we will not mention it often in the formal definition of first order logic.

Definition 2 (Substitutions)
The result of substituting a term t for a variable x in P is denoted as $P[x := t]$. The substitution is defined as follows (we assume that $x \not\equiv y$):

$$
\begin{array}{lcll}
x[x := t] & \equiv & t & \\
y[x := t] & \equiv & y & \\
(ft_1 \ldots t_n)[x := t] & \equiv & f(t_1[x := t]) \ldots (t_n[x := t]) & \\
(Pt_1 \ldots t_n)[x := t] & \equiv & P(t_1[x := t]) \ldots (t_n[x := t]) & \\
(P \wedge Q)[x := t] & \equiv & P[x := t] \wedge Q[x := t] & \\
(P \vee Q)[x := t] & \equiv & P[x := t] \vee Q[x := t] & \\
(P \Rightarrow Q)[x := t] & \equiv & P[x := t] \Rightarrow Q[x := t] & \\
(\neg P)[x := t] & \equiv & \neg(P[x := t]) & \\
(\forall x.P)[x := t] & \equiv & \forall x.P & \\
(\forall y.P)[x := t] & \equiv & \forall y.P[x := t] & \textit{Ok, because of variable convention} \\
(\exists x.P)[x := t] & \equiv & \exists x.P & \\
(\exists y.P)[x := t] & \equiv & \exists y.P[x := t] & \textit{Ok, because of variable convention}
\end{array}
$$

For the substitution equations of quantified formulas we can assume that $y \notin FV(t)$ by using the first clause of the variable convention.

Definition 3 (Closed formulas)
We say that variable x does not occur free in P if and only if $P[x := t] = P$ for all possible $t \in T$. P is called closed if no variable occurs free in P; it is called open otherwise.

$\mathcal{F}$ and $\mathcal{P}$ are the parameters of this framework. A weakness of the definitions above is that all terms are treated equally. In practice, we often want to distinguish between terms of different "types" (for instance, booleans and integers). Therefore, we will introduce a more general definition.

Definition 4 (multi-sorted first order formulas)
In addition to $\mathcal{F}$ and $\mathcal{P}$ we have a parameter Set that represents a set of basic types. With every function symbol $f \in \mathcal{F}$ with arity n, we associate a unique tuple of types $(U_1, \ldots, U_n, U)$, where $U_1, \ldots U_n$ and U are elements of Set. We denote this as $f : (U_1, \ldots U_n, U) \in \mathcal{F}$. With every predicate symbol $P \in \mathcal{P}$ with arity n, we associate a unique tuple of types $(U_1, \ldots, U_n)$, where $U_1, \ldots, U_n \in Set$. This is denoted as $P : (U_1, \ldots, U_n) \in \mathcal{P}$. Since the arity of function and predicate symbols can now be derived from its unique associated tuple of types it will no longer be stated explicitly. The set V of variables in the extended framework contains variables which each have a unique associated type U, where $U \in Set$. We assume that for every type there are infinitely many variables. The definition of the set T of typed terms is:

1. *$a : U \in T$ for every variable a with associated type U.*
2. *If $f : (U_1, \ldots, U_n, U) \in \mathcal{F}$ and $t_1 : U_1, \ldots, t_n : U_n \in T$ then $(f t_1 \ldots t_n) : U \in T$.*

The set Prop of formulas for multisorted first order logic is now defined as:

1. *If $P : (U_1, \ldots, U_n) \in \mathcal{P}$ and $t_1 : U_1, \ldots, t_n : U_n \in T$, then $(P t_1 \ldots t_n) \in Prop$.*
2. *If $P, Q \in Prop$, then $(P \wedge Q) \in Prop$, $(P \vee Q) \in Prop$ and $(P \Rightarrow Q) \in Prop$.*

3. *If $P \in Prop$, then $(\neg P) \in Prop$.*

4. *If $P \in Prop$ and $x : U \in V$, then $(\forall x : U.P) \in Prop$ and $(\exists x : U.P) \in Prop$.*

Definition 5 (Atomic formulas)
One important subset of Prop is the set of atomic formulas. *This is the set of all (propositional) formulas $Pt_1 \dots t_n \in Prop$, with $P \in \mathcal{P}$ and $t_1, \dots, t_n \in T$.*

Definition 6 (Literals)
The set of literals *is defined as the set of atomic formulas and negations of atomic formulas.*

In this thesis, we will only consider substitutions of terms for variables which have the same associated type.

This framework is more general than the first one, since the original framework can be obtained by choosing Set $\equiv \{U\}$.

5.2.1 Semantics of First Order Predicate Logic

The semantics of first order logic is such that every closed formula represents a proposition. The meaning of the proposition depends on the interpretation of the predicate symbols and function symbols the user of the logic has in mind. On the other hand, there are closed formulas that model true propositions independent of the interpretation of the user. These formulas are called tautologies. Tautologies provide self-contained information. They can be seen as legal statements that will always hold, regardless of the meaning of the predicate and function symbols.

The semantics of first order logic presented here consists of a mapping from the syntactical set of type symbols, function symbols and predicate symbols to real sets, functions and relations.

Definition 7 (Interpretations)
In this definition, we follow [dN95] page 25: Let Γ be a set of first order formulas. An interpretation I of Γ is an ordered tuple $I = (\mathcal{D}, s)$, where

- *$\mathcal{D}$ is a set of nonempty domains.*

- *s is a function which attaches*
  - *to every type symbol U in Set a domain in $\mathcal{D}$. We denote this domain by $s(U)$.*
  - *to every function symbol f occurring in Γ which has associated type $(U_1, \ldots, U_n, U)$ (and hence, arity n) a total function from the function space $s(U_1) \to \ldots \to s(U_n) \to s(U)$. We denote this function as $s(f)$.*
  - *to every predicate symbol P occurring in Γ with associated type $(U_1, \ldots, U_n)$ (and hence, arity n) a subset of $s(U_1) \times \ldots \times s(U_n)$. We denote this subset as $s(P)$. The elements of $s(P)$ are the tuples for which the relation holds. For the special symbol* **Falsum**, *we assume that $s(\textbf{Falsum}) = \emptyset$.*
  - *to every variable v with associated type U which is free in an $F \in \Gamma$ an element of $s(U)$. This element is denoted as $s(v)$.*

We extend the mapping s to attach a meaning to every term t in T:

- *If the term t is a variable v, then $s(v)$ is already defined (see above).*
- *If t has the form $ft_1 \ldots t_n$, then $s(ft_1 \ldots t_n) = s(f)(s(t_1), \ldots, s(t_n))$.*

We also define the modified mappings $s[v \mapsto d]$ where v is a variable with associated type U and d is an element of $s(U)$.The value of $s[v \mapsto d](X)$ is defined as:

- *$s(X)$ if X is a type, a function symbol, a predicate symbol or a variable different from v.*
- *d if $X = v$.*

Note that $s(\textit{Falsum}) \neq \{()\}$, the singleton set with the empty tuple as only member, which is the only other possibility for predicate symbols with arity 0. If it was, the semantics of *Falsum* would have been `t` instead of `f`.

Definition 8 (Model)
We are now ready to define a model based on the interpretation I. We denote a model of first order logic, based on interpretation $I = (\mathcal{D}, s)$ as M^I. M^I is a function from formulas to $\{\mathtt{t}, \mathtt{f}\}$ defined as:

1. *$M^I(Pt_1 \dots t_n) = \mathtt{t}$ if and only if $(s(t_1), \dots, s(t_n)) \in s(P)$.*
2. *$M^I(P \wedge Q) = \mathtt{t}$ if and only if $M^I(P) = \mathtt{t}$ and $M^I(Q) = \mathtt{t}$.*
3. *$M^I(P \vee Q) = \mathtt{t}$ if and only if $M^I(P) = \mathtt{t}$ or $M^I(Q) = \mathtt{t}$.*
4. *$M^I(P \Rightarrow Q) = \mathtt{t}$ if and only if $M^I(P) = \mathtt{f}$ or $M^I(Q) = \mathtt{t}$.*
5. *$M^I(\neg P) = \mathtt{t}$ if and only if $M^I(P) = \mathtt{f}$.*
6. *$M^I(\forall x : U.P) = \mathtt{t}$ if and only if for every $d \in s(U)$ we have $M^{I'}(P) = \mathtt{t}$ with $I' = (D, s[x \mapsto d])$.*
7. *$M^I(\exists x : U.P) = \mathtt{t}$ if and only if there exists a $d \in s(U)$ such that $M^{I'}(P) = \mathtt{t}$ with $I' = (D, s[x \mapsto d])$.*
8. *$M^I(P) = \mathtt{f}$ in all other cases.*

Note that by definition $M^I(\mathit{Falsum}) = \mathtt{f}$ for any interpretation I.

To reason about the correctness of reasoning methods like natural deduction, tableau- and resolution methods, we introduce the relation $\models$ (read as 'entails') between sets of closed formulas. The definition is as follows:

Definition 9 ($\models$)
$\Gamma \models \Delta$ if and only if for all models M^I for which $M^I(P) = \mathtt{t}$ for all $P \in \Gamma$ there is at least one $Q \in \Delta$ with $M^I(Q) = \mathtt{t}$, where Γ and Δ are sets of closed formulas.

Hence, if Γ is empty then for all models M^I at least one formula in Δ is true. This is denoted as $\models \Delta$. If, in addition, Δ contains only one element Q, this element is a tautology since $M^I(Q) = \mathtt{t}$ for all models M^I. If Δ is empty then $\Gamma \models \Delta$ is false, even if Γ is empty too, since there never is an element in Δ that is mapped to t in the interpretation I. If there are no models for which $M^I(P) = \mathtt{t}$ for every formula P in Γ, then the set Γ is called inconsistent. This is denoted by $\Gamma \models \mathit{Falsum}$. Obviously, if $\Gamma \models \mathit{Falsum}$ then $\Gamma \models Q$ for every closed formula Q. Since $M^I(\mathit{Falsum}) = \mathtt{f}$ for all models, $\mathit{Falsum} \models Q$ for every closed formula Q.

Chapter 6

Pure Type Systems

In this chapter we define and discuss Pure Type Systems (PTSs). The PTS framework is due to Terlouw and Barendregt (see [Ter89, Bar92]). The framework is used to describe several different logics in a uniform way. We use such a PTS to construct correctness proofs for imperative programs. By using a PTS as the basis of our system instead of a single fixed logic, we enable future extensions of the logical system. Also, PTSs have proved suitable for constructing interactive theorem provers like COQ [Coq97] and Yarrow [Zwa98].

6.1 Pure Type Systems (PTSs)

A PTS is specified by a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ of sets, where $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$. The elements of $\mathcal{S}$ are called sorts, the elements of $\mathcal{A}$ are called axioms and the set $\mathcal{R}$ contains (Π-formation) rules. Given a specification of a PTS, say $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, the terms, contexts and type judgment relation of the PTS are defined as follows:

Definition 10 (Terms)
Given a set V of variables, the set T of PTS terms is defined by the following abstract syntax:

$$T ::= \mathcal{S} \mid V \mid \lambda V : T.T \mid \Pi V : T.T \mid TT$$

Definition 11 (Contexts)
A context is a list of the form $x_1 : A_1, \ldots, x_n : A_n$, where $x_i \in V$ and A_i are terms as defined in definition 10 for $i = 1, \ldots, n$. The empty context is denoted as $<>$. By convention we use $\Gamma, \Delta, \ldots$ as metavariables for contexts. If $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is a context and $v \in V$, then v is called Γ-fresh if $v \notin \{x_1, \ldots, x_n\}$.

Definition 12 (β-reduction)
On the terms of PTSs, we define a reduction relation $\rightarrow_\beta \subseteq T \times T$ as the smallest relation such that

$$(\lambda x : A.b)c \rightarrow_\beta b[x := c]$$

and that is closed under

$$\begin{array}{llr} \textit{if} \quad b \rightarrow_\beta b' & \textit{then} & (\lambda x : b.c) \rightarrow_\beta (\lambda x : b'.c) \\ & & (\lambda x : a.b) \rightarrow_\beta (\lambda x : a.b'), \\ & & (\Pi x : b.c) \rightarrow_\beta (\Pi x : b'.c), \\ & & (\Pi x : a.b) \rightarrow_\beta (\Pi x : a.b'), \\ & & (bc) \rightarrow_\beta (b'c) \\ & \textit{and} & (ab) \rightarrow_\beta (ab') \end{array}$$

$\twoheadrightarrow_\beta$ denotes the reflexive and transitive closure of $\rightarrow_\beta$. $=_\beta$ denotes the symmetric, reflexive and transitive closure of $\rightarrow_\beta$.

$B =_\beta B'$ is read as "B is β-equal to B'", which means that there exists a B'' such that B and B' can both be reduced to B'' by β-reduction.

start	$<> \ \vdash s1{:}s2$	$(s1, s2) \in \mathcal{A}$
intro	$\dfrac{\Gamma \vdash A{:}s}{\Gamma, x{:}A \vdash x{:}A}$	x is Γ-fresh
weaken	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A{:}B}$	x is Γ-fresh
Π-*form*	$\dfrac{\Gamma \vdash A{:}s1 \quad \Gamma, x{:}A \vdash B{:}s2}{\Gamma \vdash (\Pi x{:}A.\ B){:}s3}$	$(s1, s2, s3) \in \mathcal{R}$
Π-*intro*	$\dfrac{\Gamma, x{:}A \vdash b{:}B \quad \Gamma \vdash (\Pi x{:}A.\ B){:}s}{\Gamma \vdash (\lambda x{:}A.\ b){:}(\Pi x{:}A.\ B)}$	
Π-*elim*	$\dfrac{\Gamma \vdash F{:}(\Pi x{:}A.\ B) \quad \Gamma \vdash a{:}A}{\Gamma \vdash \ Fa{:}B[x := a]}$	
conversion	$\dfrac{\Gamma \vdash A{:}B \quad \Gamma \vdash B'{:}s \quad B{=_\beta}B'}{\Gamma \vdash A{:}B'}$	

Figure 6.1: The type judgment derivation rules of a PTS.

Definition 13 (Type judgment relation)
The type judgment relation describes the actual PTS. A judgment always has the form $\Gamma \vdash A : B$, where A and B are terms and Γ is a context. $\Gamma \vdash A : B$ should be read as: 'A has type B in context Γ'. The type judgment relation $\vdash$ is defined by the rules in figure 6.1.

We give a brief description of each type judgment rule in figure 6.1:

start This is the only rule without premises in a PTS. It supplies, starting from the axioms in $\mathcal{A}$, basic typing judgments from which all the other typing judgments are derived.

intro Intro is used in a much more general sense than the *intro*-rule in natural deduction. In natural deduction intro allows one to add assumptions to the context. In a PTS intro allows one to add assumptions, constants (which in a PTS are equal to variables), functions and propositional variables (including predicates) to the context. This depends on the form of A. The type of the introduced item x depends on s, which is the type of the type of x.

weaken Weaken is needed to preserve existing derivations in extended contexts. It states that everything that can be derived in a certain context can also be derived in a more extended context.

Π-*form* This rule allows the construction of function types, predicates, universal quantifications, etc. The set of rules $\mathcal{R}$ of a PTS determines the ways in which Π-*form* can be used. Actually, the set $\mathcal{R}$ states which abstractions are allowed.

Π-*intro* One needs this rule to actually construct terms of a type built with the previous rule. Without this rule, we could only assume that there are terms of this type by using *intro*.

Π-*elim* Once a term with a Π-type is constructed or assumed, it can be used to create a term with a more concrete type. The Π-*elim* rule, also referred to as the application rule, instantiates the body of an abstract Π-type by substituting a term for the bound abstract variable.

conversion This rule states that we don't distinguish β-equal types. In several PTSs a term A can have type B where B can be rewritten to B' by β-reduction. In the propositions-as-types isomorphism, B and B' then represent the same propositional formula (we will come back to this in our example below) and hence, A is a proof of B' just as well as it is a proof of B. To support this switch of representation the *conversion* rule is needed.

A problem with the conversion rule is that it does not affect the term A, which makes type-checking less efficient. Suppose that a type judgment of the form $\Gamma \vdash p{:}P$ is given. In order to find out if this type judgment could be derived by using the rules in figure 6.1 (type-checking), one usually compares the syntactical structure of p with the conclusion of the rules to find the last rule that was applied. If a match is found, the type-checking results in a set of simpler type-checking questions. However, since the first premise of the conversion rule contains exactly the same proof-term as the conclusion, it is impossible to determine exclusively which rule was applied last. This problem can be eliminated by computing the β-normal form of the type before checking the proof-correctness, but this can be very inefficient.

Constructing a complete type checking algorithm that is also efficient is therefore hard.

Definition 14 (Valid (or legal) contexts)
A context is called valid or legal, if it can occur in a derivation using only the axioms of the Pure Type System. This notion is also used for extended PTSs, where additional rules may also be used.

For many PTSs one can automatically compute an entire derivation of $\Gamma \vdash p : P$ for a given context Γ, a proof term p and a formula P provided that at least one derivation exists (see e.g. [BJMP93]). Hence the proof-term can be checked for correctness (type-checked). This has two advantages:

1. Even if a large tool is used to construct a proof-term p, correctness of the proof is assured by type-checking. This algorithm is relatively simple and can be proved to be correct.

2. Communicating proofs corresponds to communicating a syntactical proof term. This proof term can then be checked by another proof system based on λ-calculus.

6.1.1 An example PTS for First-Order Predicate Logic

With this definition of PTSs, we are ready to demonstrate the propositions as types isomorphism. We consider the PTS for first order predicate logic as proposed by Berardi (presented in definition 5.4.5 of [Bar92]). The specification is:

$$\begin{array}{lcl} \mathcal{S} & = & \{*_s, *_p, *_f, \Box_s, \Box_p\} \\ \mathcal{A} & = & \{(*_s, \Box_s), (*_p, \Box_p)\} \\ \mathcal{R} & = & \{(*_s, *_s, *_f), (*_s, *_f, *_f), (*_s, \Box_p, \Box_p), (*_p, *_p, *_p), (*_s, *_p, *_p)\} \end{array}$$

The sorts $\mathcal{S}$ have the following intended meaning: A terms U of type $*_s$ corresponds to a type $U' \in Set$ as used in the semantics of multi-sorted first order logic. Terms of type $*_f$ are themselves types of functions. Terms of type $*_p$ represent formulas. Terms of type $\Box_p$ represent types of predicates. Note that since $(*_p, \Box_p) \in \mathcal{A}$, formulas are predicates with arity 0 (formulas have type $*_p$, which by this axiom is the type of predicates of arity 0).

We introduce the following shorthand: $\Gamma \vdash A : B : s$, with $s \in \mathcal{S}$ denotes that $\Gamma \vdash B : s$ and $\Gamma \vdash A : B$. Then, if we have $\Gamma \vdash A : B : *_s$, A corresponds to a term with a value in the set B. If we have $\Gamma \vdash A : B : *_f$, then A is a function with type B. If $\Gamma \vdash A : B : *_p$, then A is a proof of the formula B. If $\Gamma \vdash A : B : \Box_p$ then A is a predicate.

To make these correspondences more visible, we will use different notations for various Π-types. A term $(\Pi x : A.B)$ formed by Π-*form* with $(*_p, *_p, *_p) \in \mathcal{R}$ is denoted as $A \Rightarrow B$ (it can be proved that in this logic $x \notin FV(B)$). A term $(\Pi x : A.B)$ formed with rule $(*_s, *_p, *_p)$ is denoted as $(\forall x : A.B)$. A proof of $(\forall x : U.(Px \Rightarrow Px))$ can then be derived as:

1	$<>$	$\vdash *_s : \Box_s$	(*start*)
2	$U : *_s$	$\vdash U : *_s$	(*intro* 1)
3	$<>$	$\vdash *_p : \Box_p$	(*start*)
4	$U : *_s$	$\vdash *_p : \Box_p$	(*weaken* 1,3)
5	$U : *_s, x : U$	$\vdash *_p : \Box_p$	(*weaken* 2,4)
6	$U : *_s$	$\vdash (\Pi x : U.*_p) : \Box_p$	(Π-*form* 2,5)
7	$U : *_s, P : (\Pi x : U.*_p)$	$\vdash P : (\Pi x : U.*_p)$	(*intro* 6)
8	$U : *_s, P : (\Pi x : U.*_p)$	$\vdash U : *_s$	(*weaken* 2,6)
9	$U : *_s, P : (\Pi x : U.*_p), x : U$	$\vdash P : (\Pi x : U.*_p)$	(*weaken* 7,8)
10	$U : *_s, P : (\Pi x : U.*_p), x : U$	$\vdash x : U$	(*intro* 8)
11	$U : *_s, P : (\Pi x : U.*_p), x : U$	$\vdash Px : *_p$	(Π-*elim* 9,10)
12	$U : *_s, P : (\Pi x : U.*_p), x : U, p : Px$	$\vdash Px : *_p$	(*weaken* 11,11)
13	$U : *_s, P : (\Pi x : U.*_p), x : U, p : Px$	$\vdash p : Px$	(*intro* 11)
14	$U : *_s, P : (\Pi x : U.*_p), x : U$	$\vdash Px \Rightarrow Px : *_p$	(Π-*form* 11,12)
15	$U : *_s, P : (\Pi x : U.*_p), x : U$	$\vdash (\lambda p : Px.p) : Px \Rightarrow Px$	(Π-*intro* 13,14)
16	$U : *_s, P : (\Pi x : U.*_p)$	$\vdash (\forall x : U.(Px \Rightarrow Px)) : *_p$	(Π-*form* 8,14)
17	$U : *_s, P : (\Pi x : U.*_p)$	$\vdash (\lambda x : U.(\lambda p : Px.p)) :$ $(\forall x : U.(Px \Rightarrow Px))$	(Π-*intro* 15,16)

The example above is quite dull, but already requires a derivation of 17 lines. Equally, we could derive the type judgment:

$$\begin{aligned} &U : *_s, P : (\Pi x : U.*_p), Q : (\Pi x : U.*_p) \vdash \\ &\qquad (\lambda p : (\forall x : U.(Px \Rightarrow Qx)).(\lambda q : (\forall x : U.Px).(\lambda x : U.px(qx)))) \\ &\qquad : (\forall x : U.(Px \Rightarrow Qx)) \Rightarrow ((\forall x : U.Px) \Rightarrow (\forall x : U.Qx)) \end{aligned}$$

However, then the derivation becomes no less than 47 lines. The reason for this is the necessity to derive type correctness judgments (Π-*form*) and the step-by-step usage of weakening (see lines 3,4,5).

Since λ-calculus is a language to express mathematical functions, how can it be used to represent proofs of logical formulas? In λ-calculus the proof object $(\lambda x : U.(\lambda p : Px.p))$ is a function that returns for each element x of U a proof of $Px \Rightarrow Px$. This proof is given by $(\lambda p : Px.p)$, which in turn is a function that given a proof of Px returns a proof of Px (the same proof). Intuitively, the existence of such a function is indeed a proof of $(\forall x : U.(Px \Rightarrow Px))$: given any element a of U and any proof of Pa the function returns a proof of Pa, hence $(\forall x : U.(Px \Rightarrow Px))$.

Note that, in our example above, the set U and the unary predicate P occur explicitly in the context of the PTS. In fact the sets, functions and predicates of the logic can be expressed in PTSs by elements in the context. In logic, the functions and predicates are defined beforehand by the sets Set, $\mathcal{F}$ and $\mathcal{P}$. Explicitly stating all functions and predicates in a context allows flexible logics to be handled by proof systems based on PTSs.

However, this PTS does not model first order logic exactly. There are a few differences that are not always desirable:

1. Constants, like the natural number 0, are modeled in a context by $\Gamma_1, Nat : *_s, 0 : Nat, \Gamma_2$. Therefore they are indistinguishable from ordinary variables like the $x : U$ in line 9 of our example derivation.

2. Functions themselves have types. More precisely, a binary function f with arguments from sets A and B yielding a value from C has type $(\Pi x : A.(\Pi y : B.C))$. If f is applied to an argument $a : A$ then fa has type $(\Pi y : B.C)$, while in first order logic f applied to just one argument does not have a meaning at all. The same holds for predicates. That functions can be applied to less arguments than indicated by their arity is called currying.

3. A single proposition corresponds to several types. For instance: in context $U : *_s, P : *_p, a : U$ the term P represents a predicate of the logic with arity 0, but in this context the same predicate is represented by $(\lambda x : U.P)a$. This is why the rule *conversion* is needed: a proof $p : P$ should also be a proof of $(\lambda x : U.P)a$, since it represents the same proposition. The problem appears to be caused by the rule $(*_s, \Box_p, \Box_p)$, which allows the creation of such λ-terms. This rule is absolutely necessary, however, to construct types of predicates of arities larger than zero (See lines 6,7 in the derivation of the proof of $(\forall x{:}U.\ Px \Rightarrow Px)$ given above).

6.2 PTSs with parameters

The less desirable properties of the PTS for first order logic given in the previous part of this section can be avoided by using an extension of the PTS definition described in [Laa97]. The extension introduces parametric constants added to the terms of a PTS. A parametric constant is kept in the context and can only be used if all the required parameters are supplied at once. This corresponds to the way predicates and functions are used in first order logic. PTSs extended with parametric constants are called CPTSs.

A CPTS is specified by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P})$, where $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$ are the sorts, axioms and rules of a regular PTS and $\mathcal{P}$ is a subset of $\mathcal{S} \times \mathcal{S}$. $\mathcal{P}$ is called the set of parametric rules.

Definition 15 (Parametric Terms)
Given a set V of variables and a set C of constants, the set T_C of CPTS terms is defined by the following abstract syntax:

$$\begin{array}{lcl} T_C & ::= & \mathcal{S} \mid V \mid \lambda V : T_C.T_C \mid \Pi V : T_C.T_C \mid T_C T_C \mid C(L_C) \\ L_C & ::= & \varepsilon \mid < L_C, T_C > \end{array}$$

The lists of terms produced by L_C are usually denoted as $< A_1, \ldots, A_n >$ or $A_1, \ldots, A_n$ instead of $< \ldots << \varepsilon, A_1 >, A_2 > \ldots A_n >$.

Definition 16 (Contexts of CPTSs)
A context is a list of the form $x_1 : A_1, \ldots, x_n : A_n$, such that every A_i is a term as defined in definition 15 and either $x_i \in V$ or x_i has the form $c(y_1 : B_1, \ldots, y_m : B_m)$, where $c \in C$, $y_1, \ldots, y_m \in V$ and $B_1, \ldots, B_m$ are terms as defined in definition 15. A constant c is called Γ-fresh if it does not occur in Γ.

Definition 17 (Type judgment relation of a CPTS)
The type judgment relation of a CPTS consists of all rules of a regular PTS (see figure 6.1) and two additional rules to make use of parametric constants. Let Δ denote $x_1 : B_1, \ldots, x_n : B_n$ and Δ_i denote $x_1 : B_1, \ldots, x_{i-1} : B_{i-1}$. Then the additional rules are:

$$\textit{C-weaken}\quad \frac{\begin{array}{lll}\Gamma\vdash b{:}B & & \\ \Gamma,\Delta_i\vdash B_i{:}s_i & \textit{for } i=1,\ldots,n & (s_i,s)\in\mathcal{P}\\ \Gamma,\Delta\vdash A{:}s & & c \textit{ is } \Gamma\textit{-fresh}\end{array}}{\Gamma,c(\Delta){:}A\vdash b{:}B}$$

$$\textit{C-application}\quad \frac{\begin{array}{ll}\Gamma_1,c(\Delta){:}A,\Gamma_2\vdash b_i{:}B_i[x_j:=b_j]_{j=1}^{i-1} & \textit{for } i=1,\ldots,n\\ \Gamma_1,c(\Delta){:}A,\Gamma_2\vdash A{:}s & \textit{if } n=0\end{array}}{\Gamma_1,c(\Delta){:}A,\Gamma_2\vdash c(b_1,\ldots,b_n){:}A[x_j:=b_j]_{j=1}^{n}}$$

We give a brief description of the additional rules:

C-weaken *The* C-weaken *rule allows us to add a parametric constant to the context. In contrast to other extensions of the context this rule does not allow us to type the parametric constant itself, while the intro-rule (used for regular extensions of the context) allows the typing of every newly added item.*

C-application *Since a parametric constant itself cannot be typed in a CPTS it cannot be used with the usual* Π-elim *(sometimes called* application*) rule. The rule* C-application *allows us to use a parametric constant, but only if we supply all the required arguments at once. This corresponds to the use of functions and predicates in first order logic: these too can only be used after all the arguments have been supplied. The special premise for the case $n = 0$ is needed to assure that the context $\Gamma_1, c(\Delta) : A, \Gamma_2$ is a valid one.*

6.2.1 Properties of CPTSs

CPTSs have the usual meta-theoretical properties of Pure Type Systems. We list the most important ones below. Proofs can be found in [Laa97].

Lemma 18 (Start Lemma)
If $\Gamma\vdash A{:}B$ then $\Gamma\vdash s_1{:}s_2$ for any $(s_1,s_2)\in\mathcal{A}$.

Lemma 19 (Weakening Lemma)
If $\Gamma_1\vdash A{:}B$ and Γ_1,Γ_2 is a valid context, then $\Gamma_1,\Gamma_2\vdash A{:}B$.

Lemma 20 (Substitution Lemma)
If $\Gamma, x : A, \Delta \vdash B{:}C$ *and* $\Gamma \vdash D{:}A$ *then* $\Gamma, \Delta[x := D] \vdash B[x := D]{:}C[x := D]$.

Lemma 21 (Generation Lemma)

1. *If* $\Gamma \vdash s{:}C$ *for an* $s \in \mathcal{S}$ *then there is* $s' \in \mathcal{S}$ *such that* $C =_\beta s'$ *and* $(s, s') \in \mathcal{A}$*;*
2. *If* $\Gamma \vdash x{:}C$ *then there is* $s \in \mathcal{S}$ *and* $B =_\beta C$ *such that* $\Gamma \vdash B{:}s$ *and* $(x : B) \in \Gamma$*;*
3. *If* $\Gamma \vdash (\Pi x{:}A.\ B){:}C$ *then there is* $(s_1, s_2, s_3) \in \mathcal{R}$ *such that* $\Gamma \vdash A{:}s_1$, $\Gamma, x : A \vdash B{:}s_2$ *and* $C =_\beta s_3$*;*
4. *If* $\Gamma \vdash (\lambda x{:}A.\ b){:}C$ *then there is* $s \in \mathcal{S}$ *and* B *such that* $\Gamma, x : A \vdash b{:}B$*;* $\Gamma \vdash (\Pi x{:}A.\ B){:}s$*; and* $C =_\beta (\Pi x{:}A.\ B)$*;*
5. *If* $\Gamma \vdash Fa{:}C$ *then there are* A, B *such that* $C =_\beta B[x := a]$, $\Gamma \vdash a{:}A$ *and* $\Gamma \vdash F{:}(\Pi x{:}A.\ B)$*;*
6. *If* $\Gamma \vdash c(b_1, \ldots, b_n){:}D$ *then there exist* s, $\Delta \equiv x_1 : B_1, \ldots, x_n : B_n$ *and* A *such that* $D =_\beta A[x_i := b_i]_{i=1}^n$, *and* $\Gamma \vdash b_i{:}B_i[x_j := b_j]_{j=1}^{i-1}$. *Moreover,* $\Gamma = \Gamma_1, c(\Delta){:}A, \Gamma_2$ *and* $\Gamma_1, \Delta \vdash A{:}s$. *Finally, there are* $s_i \in \mathcal{S}$ *such that* $\Gamma, \Delta_i \vdash B_i{:}s_i$ *and* $(s_i, s) \in \mathcal{P}$.

Lemma 22 (Correctness of Types)
If $\Gamma \vdash A{:}B$ *then for some* $s \in \mathcal{S}$ *we have* $B \equiv s$ *or* $\Gamma \vdash B{:}s$.

Lemma 23 (Unicity of Types)
If $\Gamma \vdash A{:}B_1$ *and* $\Gamma \vdash A{:}B_2$ *then* $B_1 =_\beta B_2$.

Lemma 24 (Subject Reduction)
If $\Gamma \vdash A{:}B$ *and* $A {\rightarrow_\beta} A'$ *then* $\Gamma \vdash A'{:}B$.

Lemma 25 (Permutation Lemma)

- *If* x *does not occur free in* B*, then* $\Gamma_1, x : A, y : B, \Gamma_2 \vdash P{:}Q$ *if and only if* $\Gamma_1, y : B, x : A, \Gamma_2 \vdash P{:}Q$*;*
- *If* c *does not occur in* B*, then* $\Gamma_1, c(\Delta) : A, y : B, \Gamma_2 \vdash P{:}Q$ *if and only if* $\Gamma_1, y : B, c(\Delta) : A, \Gamma_2 \vdash P{:}Q$*;*

- *If x does not occur free in Δ or B, then* $\Gamma_1, x : A, c(\Delta) : B, \Gamma_2 \vdash P{:}Q$ *if and only if* $\Gamma_1, c(\Delta) : B, x : A, \Gamma_2 \vdash P{:}Q$*;*

- *If c does not occur in Δ' or B, then* $\Gamma_1, c(\Delta) : A, c'(\Delta') : B, \Gamma_2 \vdash P{:}Q$ *if and only if* $\Gamma_1, c'(\Delta') : B, c(\Delta) : A, \Gamma_2 \vdash P{:}Q$*.*

Lemma 26 (Type inference)
Let Γ be a context and A be a term. One can compute a term B, such that $\Gamma \vdash A{:}B$ if such B exists.

6.3 PTSs with definitions

As we have seen in the example, λ-terms can easily become very large. Sometimes, their size can be cut down by a λ-abstraction/λ-application pair. For instance

$$\begin{array}{ll} & (\lambda P{:}*_p\,.\,P{\Rightarrow}P)((A{\Rightarrow}B){\Rightarrow}(B{\Rightarrow}C){\Rightarrow}(C{\Rightarrow}D){\Rightarrow}(A{\Rightarrow}D)) \\ =_\beta & \\ & ((A{\Rightarrow}B){\Rightarrow}(B{\Rightarrow}C){\Rightarrow}(C{\Rightarrow}D){\Rightarrow}(A{\Rightarrow}D)){\Rightarrow} \\ & \qquad ((A{\Rightarrow}B){\Rightarrow}(B{\Rightarrow}C){\Rightarrow}(C{\Rightarrow}D){\Rightarrow}(A{\Rightarrow}D)) \end{array}$$

but the first notation is shorter. Unfortunately, using the first notation is not always possible. In the PTS for first order logic for instance, the first term is not typable. Building the λ-abstraction would require the rule $(\Box_p, \Box_p, \Box_p)$, which is not an element of $\mathcal{R}$.

To avoid this problem we extend PTSs with a definition mechanism introduced by [Pol94]. The simplest way to understand the definition mechanism is to consider definitions to be shorthand notations for larger terms. Since we can only introduce definitions for terms that could already be constructed in the PTS, the definitions mechanism does not extend the expressive power of the logic.

This definition mechanism can easily be combined with the parameter mechanism of the previous section. In [Laa97] this combination leads to parametric definitions, but in this thesis only plain definitions are used. Since plain definitions are entirely independent of the parameter mechanism, we will present only regular PTSs extended with definitions, called DPTSs. PTSs with both extensions are called CDPTSs.

Definition 27 (Definition Terms)
Given a set V of variables, the set T_D of DPTS terms is defined by the following abstract syntax:

$$T_D ::= \mathcal{S} \mid V \mid \lambda V : T_D.T_D \mid \Pi V : T_D.T_D \mid T_D T_D \mid V = T_D : T_D \ in \ T_D$$

In a term $x = a : A \ in \ b$ the occurrences of variable x in b are bound by the definition $a{:}A$. Such a term is read as "x is locally defined as a of type A in b". Hence, x is called a local definition in b.

Definition 28 (Contexts of DPTSs)
A context C is a list of the form $x_1 : A_1, \dots, x_n : A_n$, such that every A_i is a term as defined in definition 27 and either $x_i \in V$ or x_i has the form $y = B$, where $y \in V$ and B is a term as defined in definition 27. Items $y = B : A$ in a context are called global definitions.

Definition 29 (δ-reduction)
On the terms and contexts of DPTSs a reduction relation

$$\_ \vdash \_ \rightarrow_\delta \_ \subset Contexts \times T_D \times T_D$$

is defined as the smallest relation such that

$$\begin{array}{ll} \Gamma_1, x = a : A, \Gamma_2 \vdash x \rightarrow_\delta a & \\ \Gamma \vdash (x = a : A \ in \ b) \rightarrow_\delta b & \text{if } x \notin FV(b) \end{array}$$

and that is closed under the following compatibility rules

$$\begin{array}{llll} \text{if} & \Gamma, x = a : A \vdash b \rightarrow_\delta b' & \text{then} & \Gamma \vdash (x = a : A \ in \ b) \rightarrow_\delta (x = a : A \ in \ b') \\ \text{if} & \Gamma, x : A \vdash b \rightarrow_\delta b' & \text{then} & \Gamma \vdash (\Pi x{:}A.\ b) \rightarrow_\delta (\Pi x{:}A.\ b') \\ & & \text{and} & \Gamma \vdash (\lambda x{:}A.\ b) \rightarrow_\delta (\lambda x{:}A.\ b') \\ \text{if} & \Gamma \vdash a \rightarrow_\delta a' & \text{then} & \Gamma \vdash (x = a : A \ in \ b) \rightarrow_\delta (x = a' : A \ in \ b), \\ & & & \Gamma \vdash (x = A : a \ in \ b) \rightarrow_\delta (x = A : a' \ in \ b), \\ & & & \Gamma \vdash (ab) \rightarrow_\delta (a'b), \\ & & & \Gamma \vdash (ba) \rightarrow_\delta (ba'), \\ & & & \Gamma \vdash (\Pi x{:}a.\ b) \rightarrow_\delta (\Pi x{:}a'.\ b) \\ & & \text{and} & \Gamma \vdash (\lambda x{:}a.\ b) \rightarrow_\delta (\lambda x{:}a'.\ b) \end{array}$$

We write $\Gamma \vdash a \rightarrow_{\beta\delta} b$ if $a \rightarrow_\beta b$ or $\Gamma \vdash a \rightarrow_\delta b$. Furthermore, $\_ \vdash \_ \twoheadrightarrow_{\beta\delta} \_$ and $\_ \vdash \_ \twoheadrightarrow_\delta \_$ are the reflexive and transitive closures of $\_ \vdash \_ \rightarrow_{\beta\delta} \_$ and $\_ \vdash \_ \rightarrow_\delta \_$ respectively. $\_ \vdash \_ =_{\beta\delta} \_$ and $\_ \vdash \_ =_\delta \_$ are the symmetric, reflexive and transitive closures of $\_ \vdash \_ \rightarrow_{\beta\delta} \_$ and $\_ \vdash \_ \rightarrow_\delta \_$ respectively.

In [Pol94] it is proved that $\rightarrow_\delta$ yields a terminating rewriting system that has unique normal forms and has the Church-Rosser property.

Definition 30 (Type judgment relation of a DPTS)
The type judgment relation of a DPTS consists of all rules of a regular PTS (see figure 6.1) and the following additional rules to make use of definitions ($s \in \mathcal{S}$):

D-intro $$\frac{\Gamma \vdash a{:}A}{\Gamma, x = a : A \vdash x{:}A} \quad x \textit{ is } \Gamma\textit{-fresh}$$

D-weaken $$\frac{\Gamma \vdash b{:}B \quad \Gamma \vdash a{:}A}{\Gamma, x = a : A \vdash b{:}B} \quad x \textit{ is } \Gamma\textit{-fresh}$$

δ-form $$\frac{\Gamma, x = a : A \vdash B{:}s}{\Gamma \vdash (x = a : A \textit{ in } B){:}s}$$

δ-intro $$\frac{\Gamma, x = a : A \vdash b{:}B \quad \Gamma \vdash (x = a : A \textit{ in } B){:}s}{\Gamma \vdash (x = a : A \textit{ in } b){:}(x = a : A \textit{ in } B)}$$

βδ-conversion $$\frac{\Gamma \vdash b{:}B \quad \Gamma \vdash B'{:}s \quad \Gamma \vdash B =_{\beta\delta} B'}{\Gamma \vdash b{:}B'}$$

Again we give a brief description of the additional rules:

D-intro *D-intro* enables the user of the logic to introduce a global definition into the context.

D-weaken This additional weaken-rule states that a type judgment that holds in a context Γ, also holds in the same context extended with a definition.

δ-form With *δ-form* the user can replace a global definition by a local definition in terms that are inhabitants of sorts. This can be compared to the *Π-form* rule that allows the formation of types for λ-abstractions. However, this rule does not depend on the set $\mathcal{R}$ and hence, any definition can be introduced regardless of the logical abstractions that are allowed.

δ-intro Like *Π-intro*, the *δ-intro* rule allows the construction of terms which have a type containing a (local) definition. That is, in $x = a : A$ in B,

'=' is a binder, which binds all occurrences of x in B. Every x occurring in B is a shorthand for term $a : A$. In contrast to λ-terms that have Π-types, the terms and types of definition constructs use the same binder.

$\beta\delta$-conversion This rule serves the same purpose as *conversion*, namely that it eliminates the differences between terms that are $\beta\delta$-equal. In fact, since for any terms B and B' and any context Γ we have that $B =_\beta B'$ implies $\Gamma \vdash B =_{\beta\delta} B'$ the rule *conversion* becomes superfluous.

6.4 Theorems in PTSs

When constructing larger proofs, one usually uses lemma's and theorems. These lemma's and theorems have typically been proved earlier and are used on several occasions for differently named predicates and propositions. In a formal framework, like PTSs, this re-use of theorems for different predicates must be made explicit, since a proof of $P \wedge Q \Rightarrow P \vee Q$ is different from a proof of $A \wedge B \Rightarrow A \vee B$.

In many PTSs it is sufficient to abstract away over the actual propositions; i.e. given a type judgment

$$\Gamma, P : *_p, Q : *_p \vdash p{:}P \wedge Q \Rightarrow P \vee Q$$

we construct the judgment

$$\Gamma \vdash (\lambda P{:}\, *_p\,.\; \lambda Q{:}\, *_p\,.\; p){:}(\Pi P{:}\, *_p\,.\; (\Pi Q{:}\, *_p\,.\; P \wedge Q \Rightarrow P \vee Q)).$$

To obtain the original proof we apply the λ-term to P and Q respectively; but applying the same λ-term to A and B yields a proof of $A \wedge B \Rightarrow A \vee B$.

Unfortunately, the construction of such a λ-term is not always possible: To create the Π-type $\Pi Q{:}\, *_p\,.\; P \wedge Q \Rightarrow P \vee Q$ we need the PTS-rule $(\Box_p, *_p, *_p)$ in $\mathcal{R}$, but not all PTSs have this rule, notably the system for first order logic.

Another solution would be to use parametric definitions, i.e. a combination of the definition mechanism and parametric constants as described in [Laa97]. In a local context, the definition is then considered to be a parametric constant and hence, it is not typeable unless the required arguments are provided. Since the parametric definition is not typeable itself, the PTS does not have to be extended.

However, since parametric constants are not typeable either, parametric definitions cannot abstract away over functions and predicates and hence, cannot be used to define theorems about functions and predicates.

For instance, consider the following theorem: Let U be a set and let P be a predicate on pairs of U, then if there exists an x in U such that for all y in U the predicate $P(x, y)$ holds, then for all q in U there exists a p in U such that $P(p, q)$ holds. In a CPTS this is stated (and proved) as follows:

$$\begin{aligned} U : *_s, P(x : U, y : U) : *_p \vdash t{:}&(\exists x{:}U.\ \forall y{:}U.\ P(x, y)) \\ &\Rightarrow (\forall q{:}U.\ \exists p{:}U.\ P(p, q)) \end{aligned} \tag{6.1}$$

where t is the required proof term. However, even in a higher order CPTS one cannot abstract away over P, since it is a constant. Also, since P is not typeable itself, it is not possible to define a parametric constant (or definition) that takes P as an argument. Changing the rules to allow such abstractions showed to allow ill-typed sub-terms in unused definitions. Even though this did not lead to invalid systems, the resulting systems lost a few aesthetically nice properties.

A third solution is not to capture the concept of theorems within the formalism at all, but to add the usage of theorems as a feature to the system. Given a proved proposition, the system must then be able to construct a proof of a similar proposition, by substituting correct sub-proofs in the original proof. The original proof then serves as a template for constructing similar proofs. For instance, if in the example above the proof p has the form

$$\lambda H{:}(\exists x{:}U.\ \forall y{:}U.\ P(x, y)).\ q(H),$$

then the system should be able to construct a proof

$$\lambda H'{:}(\exists x'{:}U'.\ \forall y'{:}U'.\ P'(x', y')).\ q'(H')$$

in an attempt to prove

$$(\exists x'{:}U'.\ \forall y'{:}U'.\ P'(x', y')) \Rightarrow (\forall q'{:}U'.\ \exists p'{:}U'.\ P'(p', q')).$$

This last solution does not alter the formal system in any way, since for every proposition to be proved a lambda term is constructed using only the derivation rules given by the formalism. Yet, to the user the convenience of theorems is available when constructing larger proofs.

We will now give a meta-theorem about CPTSs, justifying the way in which our system will support theorems:

Theorem 31 *Let $\Gamma = x_1 : A_1, \ldots, x_n : A_n \vdash p{:}P$ be a valid type-judgment in a CPTS. Furthermore, let Δ be a legal context and t_i be terms such that for $i \in \{1, \ldots, n\}$ we have valid type-judgments $\Delta \vdash t_i{:}A_i[x_j := t_j]_{j=1}^{i-1}$. Without loss of generality we assume $\{x_1, \ldots, x_n\} \cap FV(\Delta) = \emptyset$. Then $\Delta \vdash p[x_j := t_j]_{j=1}^{n}{:}P[x_j := t_j]_{j=1}^{n}$.*

Proof sketch: From $\Gamma \vdash p{:}P$ and the weakening lemma (lemma 19) we get $\Gamma, \Delta \vdash p{:}P$. Using the permutation lemma (lemma 25) repeatedly we get $\Delta, \Gamma \vdash p{:}P$. Using the substitution lemma (lemma 20) repeatedly we get respectively:

$$\begin{array}{l} \Delta, x_1 : A_1, \ldots, x_n : A_n \vdash p{:}P \\ \Delta, (x_2 : A_2, \ldots, x_n : A_n)[x_1 := t_1] \vdash p[x_1 := t_1]{:}P[x_1 := t_1] \\ \vdots \\ \Delta \vdash p[x_j := t_j]_{j=1}^{n}{:}P[x_j := t_j]_{j=1}^{n} \end{array}$$

□

If one wants to use the theorem above to apply theorems within a theorem prover, it is recommended to store the proof-template (i.e. the type judgment $\Gamma \vdash p{:}P$) with as little context as possible. This way, less terms t_i are needed to apply the theorem. Also, it is recommended that the system searches for the terms t_i automatically, either by using an automated theorem prover or by using heuristics (like searching for variables of the correct type within Δ). If a certain required term, say t_i, cannot be found by the system itself, it is also possible to add $\Delta \vdash ?_i{:}A[x_j := t_j]_{j=1}^{i-1}$ as a goal to the proof session. The user then has to construct the term for herself.

6.5 $\lambda P-$: A CPTS for First Order Logic

We are now ready to introduce the system $\lambda P-$. $\lambda P-$ is a CPTS that exactly models many sorted first order predicate logic.

Definition 32 ($\lambda P-$)
$\lambda P-$ is the CPTS specified by:

$$\begin{array}{rcl} \mathcal{S} & = & \{*_s, *_p, \Box_s, \Box_p\} \\ \mathcal{A} & = & \{(*_s, \Box_s), (*_p, \Box_p)\} \\ \mathcal{R} & = & \{(*_p, *_p, *_p), (*_s, *_p, *_p)\} \\ \mathcal{P} & = & \{(*_s, *_s), (*_s, \Box_p)\} \end{array}$$

Note that the sort $*_f$, used by Berardi to model function types, is not present in $\lambda P-$. Also, the only rules in $\lambda P-$ are those corresponding to implication and universal quantification.

Functions and predicates are now added to the context using the new rule *C-weaken*, using parametric rule $(*_s, *_s)$ for functions and $(*_s, \Box_p)$ for predicates. A function or a predicate can only be used to form a proposition using the rule *C-application*. For instance, a function of arity 2 can only be used when it is applied to 2 arguments at once.

Essentially the propositions-as-types isomorphism and the intended meanings of the sorts of this system are equal to those of the regular PTS of Berardi. However, $\lambda P-$ corresponds more closely to first order logic:

1. Constants are now modeled by a parametric constant with zero parameters. The natural number 0 is then modeled in a context as $\Gamma_1, Nat : *_s, 0() : Nat, \Gamma_2$. Since the 0 is now a constant from C, it cannot be confused with a variable from V, since it is not possible to build a term like $(\lambda 0() : Nat.X)$.
2. Functions themselves do not have types. A binary function f with arguments from sets A and B yielding a value from C occurs in the context as $f(x : A, y : B) : C$. Since f is a parametric constant with 2 arguments it cannot be applied to a single argument $a : A$. The same holds for predicates.
3. A single proposition corresponds to a single type. The rule $(*_s, \Box_p, \Box_p)$ allowing the typing of lambda terms representing predicates is no longer available. Therefore, a predicate P is no longer represented by $(\lambda x : U.P)a$, where U corresponds to a set of first order logic and $a : U$. The rule *conversion* is no longer needed, allowing a simpler and faster implementation.

We will prove that the conversion rule is superfluous in $\lambda P-$. For this, we formally define what a 'type' is and then prove that all types in $\lambda P-$ are in β-normal form. These proofs can also be found in [LF00].

Definition 33 *Let Γ be a context. A is a* type *in Γ if $A \in \mathcal{S}$ or there is $s \in \mathcal{S}$ such that $\Gamma \vdash A : s$.*

Correctness of Types (Lemma 22) indicates that B is a type if $\Gamma \vdash A : B$. In the proof of the theorem 35 we will need the following lemma:

Lemma 34 *If* $\Gamma \vdash P : Q$ *and* $\Gamma \vdash Q : *_s$ *then* P *is in* β*-normal form.*

This lemma shows that the terms that represent objects are always in β-normal form.

Proof: Induction on the structure of P. The cases $P \in V$ and $P \in \mathcal{S}$ are trivial. Moreover, using the lemmas above it can be shown that the cases $P \equiv \Pi x{:}P_1.P_2$, $P \equiv \lambda x{:}P_1.P_2$ and $P \equiv P_1P_2$ cannot occur (see the proof of Lemma 6.82 in [Laa97]). We focus on the most important case: $P \equiv c(b_1, \ldots, b_n)$. It suffices to prove that the b_i are in β-normal form. By the Generation Lemma 21, there are $s, s_1, \ldots, s_n$, Δ and A such that $Q =_\beta A[x_i{:=}b_i]_{i=1}^n$, $\Gamma \vdash b_i : B_i[x_j{:=}b_j]_{j=1}^{i-1}$, $\Gamma \equiv (\Gamma_1, c(\Delta){:}A, \Gamma_2)$ $\Gamma_1, \Delta \vdash A : s$, $\Gamma_1, \Delta_i \vdash B_i{:}s_i$, and $(s_i, s) \in \mathcal{R}$ for all i.[1] By the Substitution Lemma 20, $\Gamma_1 \vdash A[x_i{:=}b_i]_{i=1}^n : s$, so by the Weakening Lemma 19, $\Gamma \vdash A[x_i{:=}b_i]_{i=1}^n : s$. As $Q =_\beta A[x_i{:=}b_i]_{i=1}^n$, we have by Subject Reduction 24 and Unicity of Types 23 that $s \equiv *_s$. As $(s_i, s) \in \mathcal{R}$, we have that for all i, $s_i \equiv *_s$. Using again the Substitution and Weakening lemmas, we find $\Gamma \vdash B_i[x_j{:=}b_j]_{j=1}^{i-1} : *_s[x_j{:=}b_j]_{j=1}^{i-1}$, which means that $\Gamma \vdash b_i : B_i[x_j{:=}b_j]_{j=1}^{i-1} : *_s$. By the induction hypothesis, b_i is in β-normal form. $\Box$

Now we prove the most important result for $\lambda P-$:

Theorem 35 *If* P *is a type in a context* Γ *then* P *is in* β*-normal form.*

Proof: Since P is a type, we have two cases (definition 33). The case $P \equiv s$ is trivial. So assume $\Gamma \vdash P : s$ for an $s \in \mathcal{S}$. Use induction on the structure of P.

P can be of the form 1. s, 2. x, 3. $\Pi x : A.B$, 4. $\lambda x : A.b$, 5. Fa, 6. $c(b_1, \ldots, b_n)$ (cf. Generation Lemma 21). Cases 1 and 2 are trivial. Case 3 follows almost immediately from the induction hypothesis. Case 4 can be shown to be impossible.

Case 5 is also impossible: by Generation Lemma, there exist R_1 and R_2, such that $\Gamma \vdash F : (\Pi x : R_1.R_2)$ and $\Gamma \vdash a : R_1$ and $s =_\beta R_2[x := a]$. Hence, by Correctness of Types and the Generation Lemma $\Gamma, x : R_1 \vdash R_2 : *_p$. By Substitution Lemma, $\Gamma \vdash R_2[x := a] : *_p$, hence by Subject Reduction $\Gamma \vdash s : *_p$ and hence $(s, *_p)$ should be an axiom, which it is not. We conclude that Case 5 does not occur.

[1]We write $\Delta \equiv x_1{:}B_1, \ldots, x_n{:}B_n$, and $\Delta_i \equiv x_1{:}B_1, \ldots, x_{i-1}{:}B_{i-1}$.

Finally, we focus on the case $P \equiv c(b_1, \dots, b_n)$. It suffices to prove that the b_is are in β-normal form. Determine with the Generation Lemma 21 (case 6) s', Δ and A such that $s =_\beta A[x_i{:=}b_i]_{i=1}^n$, $\Gamma \vdash b_i : B_i[x_j{:=}b_j]_{j=1}^{i-1}$, $\Gamma \equiv \langle \Gamma_1, c(\Delta) : A, \Gamma_2 \rangle$, $\Gamma_1, \Delta \vdash A : s'$, and determine $s_1, \dots, s_n \in \mathcal{S}$ such that $\Gamma_1, \Delta_i \vdash B_i : s_i$ and $(s_i, s') \in \mathcal{R}$. By Substitution Lemma, $\Gamma_1 \vdash A[x_i{:=}b_i]_{i=1}^n : s'$ and by Weakening Lemma, $\Gamma \vdash A[x_i := b_i]_{i=1}^n : s'$. so by Subject Reduction, $\Gamma \vdash s : s'$. This means that either $s \equiv *_s$ and $s' \equiv \Box_s$, or $s \equiv *_p$ and $s' \equiv \Box_p$. As (s_i, s') must be an element of $\mathcal{R}$, the only possibility for s' is that $s' \equiv \Box_p$ and, moreover, $s_i \equiv *_s$. Notice that from $\Gamma_1, \Delta_i \vdash B_i : s_i$, it follows by Substitution Lemma and Weakening Lemma that $\Gamma \vdash B_i[x_j := b_j]_{j=1}^{i-1} : s_i$, hence $\Gamma \vdash b_i : B_i[x_j{:=}b_j]_{j=1}^{i-1} : s_i$. So by Lemma 34, b_i is in β-normal form. $\Box$

So far, we were considering *minimal* first order logic with only implication and universal quantification. To model negation, conjunction, disjunction and existential quantification we would need a more powerful PTS allowing higher order constructs. However, this would destroy our close correspondence with first order logic. Another possibility is to further extend the abstract syntax of λ-terms and adding more rules to the type judgment relation. These extended λ-terms can then easily be translated into regular λ-terms of a PTS allowing higher order logic. However, our proof system itself then keeps its close correspondence to first order logic. Therefore, we will choose the second alternative: we extend the abstract syntax of $\lambda P-$, but only in a limited way. We will add a few rules needed for classical logic. but we use classically equivalent formulas to encode negation, conjunction, disjunction and existential quantification.

6.5.1 PTS extensions for $\lambda P-$

In order to model full first order predicate logic, the abstract syntax of $\lambda P-$ is extended to:

$$\begin{array}{lll} T_{P-} & ::= & \mathcal{S} \mid V \mid \lambda V : T_{P-}.T_{P-} \mid \Pi V : T_{P-}.T_{P-} \mid T_{P-}T_{P-} \mid C(L_{P-}) \\ & & \mid \perp \mid \textbf{classic } T_{P-}\ T_{P-} \\ L_{P-} & ::= & \varepsilon \mid < L_{P-}, T_{P-} > \end{array}$$

Formulas of first order predicate logic are coded in $\lambda P-$ as given in figure 6.2. These codings are correct, since the following logical formula are equivalent

$$\begin{array}{lll}
\mathit{cod}(\mathit{Falsum}) & = \bot & \\
\mathit{cod}(A) & = A \text{ if } A \text{ is an atomic formula different from } \mathit{Falsum} & \\
\mathit{cod}(\neg P) & = \Pi h{:}\mathit{cod}(P).\ \ \bot & \\
\mathit{cod}(P \Rightarrow Q) & = \Pi h{:}\mathit{cod}(P).\ \mathit{cod}(Q) & \\
\mathit{cod}(P \vee Q) & = \mathit{cod}(\neg P \Rightarrow Q) & \\
 & = (\Pi h'{:}(\Pi h{:}\mathit{cod}(P).\ \ \bot).\ \mathit{cod}(Q)) & \\
\mathit{cod}(P \wedge Q) & = \mathit{cod}(\neg(P \Rightarrow \neg Q)) & \\
 & = (\Pi h''{:}(\Pi h'{:}\mathit{cod}(P).\ (\Pi h{:}\mathit{cod}(Q).\ \ \bot)).\ \ \bot) & \\
\mathit{cod}(\forall x{:}U.\ P) & = (\Pi x{:}U.\ \mathit{cod}(P)) & \\
\mathit{cod}(\exists x{:}U.\ P) & = \mathit{cod}(\neg(\forall x{:}U.\ \neg P)) & \\
 & = (\Pi h''{:}(\Pi x{:}U.\ (\Pi h{:}\mathit{cod}(P).\ \ \bot)).\ \ \bot) &
\end{array}$$

Figure 6.2: Encoding first-order predicate logic in $\lambda P-$.

(they hold in the same models):

$\neg P$	is equivalent with	$P \Rightarrow \mathit{Falsum}$
$P \vee Q$	is equivalent with	$\neg P \Rightarrow Q$
$P \wedge Q$	is equivalent with	$\neg(P \Rightarrow \neg Q)$
$\exists x : U.P$	is equivalent with	$\neg(\forall x : U.\neg P)$

Also, the type judgment relation is extended by the following three rules:

$\bot$-*form* $$<>\ \vdash\ \bot :*_p$$

falsum $$\frac{\Gamma \vdash p{:}\ \bot \quad \Gamma \vdash P{:}*_p}{\Gamma \vdash p\ P{:}P}$$

classic $$\frac{\Gamma \vdash P{:}*_p \quad \Gamma \vdash p{:}\Pi h'{:}(\Pi h{:}P.\ \ \bot).\ \ \bot}{\Gamma \vdash \textbf{classic}\ P\ p{:}P}$$

We briefly comment on these rules:

$\bot$-*form* This axiom states that the special symbol $\bot$ (read as bottom) represents a proposition. $\bot$ represents the *Falsum* of first order predicate logic.

falsum This rule states that if $\bot$ holds and P is a proposition, then P holds.

classic If the second premise of the rule is read as $\Gamma \vdash p{:}\neg(\neg P)$, it is easy to see that this rule claims that P holds if $\neg P$ does not hold. Hence, this rule allows classical reasoning.

The *falsum* rule can be eliminated by using the derived rule

$$\frac{\Gamma \vdash p{:}\perp \quad \Gamma \vdash P{:}*_p}{\Gamma \vdash \textbf{classic } P\ (\lambda x : \neg P.p){:}P},$$

instead. However, we do not choose to do so, since it is unnatural to replace the *falsum* rule, which can be used in constructive proofs, by a construct that depends on the *classic* rule, which is not allowed in constructive proofs. Also, the *falsum* rule can be translated into other PTSs more easily. In higher order logics $\perp$ is usually defined as $\perp = \Pi P : *_p.P$ and hence, contradictions behave exactly as indicated by the *falsum* rule.

Except for the extensions for propositional constructs, we also need a context containing the set-, function- and predicate symbols of the logic. This context is defined as follows:

Definition 36 **($\Gamma_{\mathcal{L}}$)**
Let $\mathcal{L}$ be a logic with set symbols $U_1, \ldots, U_k$, function symbols $f_1, \ldots, f_p$ and predicate symbols $P_1, \ldots, P_n$. Furthermore, let $V_{i,j}$ denote the set symbol representing the type of the j'th argument of function f_i and let V_i denote the set symbol representing the type of the result of function f_i. Finally, let $T_{i,j}$ denote the set symbol corresponding to the type of the j'th argument of predicate P_i. Then the context $\Gamma_{\mathcal{L}}$, modeling this first order logic in $\lambda P-$, is defined as:

$$\begin{array}{l}U_1 : *_s, \ldots, U_k : *_s,\\ f_1(x_1 : V_{1,1}, \ldots, x_{s_1} : V_{1,s_1}) : V_1, \ldots, f_p(x_1 : V_{p,1}, \ldots, x_{s_p} : V_{p,s_p}) : V_p,\\ P_1(x_1 : T_{1,1}, \ldots, x_{r_1} : T_{1,r_1}) : *_p, \ldots, P_n(x_1 : T_{n,1}, \ldots, x_{r_n} : T_{n,r_n}) : *_p\end{array}$$

s_i and r_j are the arities of f_i and P_j respectively.

The close correspondence of logic $\mathcal{L}$ to $\lambda P-$ with context $\Gamma_{\mathcal{L}}$ is given by the following theorems:

Theorem 37 $\Gamma_{\mathcal{L}} \vdash U : *_s$ *if and only if U is a set symbol of $\mathcal{L}$.*

Theorem 38 *For any set symbol U of $\mathcal{L}$ we have $\Gamma_{\mathcal{L}} \vdash t : U$ if and only if t is a term in $\mathcal{L}$ whose type is represented by set symbol U.*

Theorem 39 $\Gamma_{\mathcal{L}} \vdash P : *_p$ *if and only if* P *is a proposition of* $\mathcal{L}$.

Theorem 40 *For any proposition* P *of* $\mathcal{L}$ *we have* $\Gamma_{\mathcal{L}} \vdash p : P$ *for some term* p *if and only if* $\models_{\mathcal{L}} P$.

Theorems 37 through 39 are proved by induction on the term structure. The completeness part of theorem 40 follows from the algorithm we present in chapter 9: the method of tableaux is complete and every closed tableau can be converted to a proof in $\lambda P-$ in context $\Gamma_{\mathcal{L}}$.

The converse is also true: if Γ is a valid context of $\lambda P-$, then there exists a logic $\mathcal{L}$ such that theorems 37 through 39 with $\Gamma_{\mathcal{L}}$ replaced by Γ hold. Hence, $\lambda P-$ has a one-to-one correspondence with many-sorted first-order predicate logic (for a proof see [LF00]).

Since propositions in $\lambda P-$ and formulas in first order logic have a one-to-one correspondence, we will from now on also use the logical notations to denote types in $\lambda P-$. I.e. we will write $\neg\neg B$ instead of $(\Pi H{:}(\Pi H'{:}B.\ \bot).\ \bot)$.

Even though it is possible to prove every valid proposition with $\lambda P-$, it is not convenient. Logicians familiar with first order logic will want to use the usual introduction and elimination rules for $\wedge$, $\vee$ and $\exists$. Therefore we will introduce a number of derived rules and their corresponding shorthands (these shorthands will be used in this thesis in chapter 9. The implementation uses the λ-terms directly, not the shorthand notation.):

$$\wedge\text{-}intro \quad \frac{\Gamma \vdash p{:}P \quad \Gamma \vdash q{:}Q}{\Gamma \vdash (p,q){:}P \wedge Q} \ ,$$

where (p,q) denotes $(\lambda H{:}P \Rightarrow (Q \Rightarrow\bot).\ Hpq)$, which has as type the coding of $P \wedge Q$.

$$\wedge\text{-}elim_1 \quad \frac{\Gamma \vdash p{:}P \wedge Q}{\Gamma \vdash \pi_1(p){:}P} \ ,$$

where $\pi_1(p)$ denotes **classic** $P\ (\lambda H{:}P \Rightarrow\bot\ .\ p(\lambda H'{:}P.\ HH'(Q \Rightarrow\bot)))$, which has as type P.

$$\wedge\text{-}elim_2 \quad \frac{\Gamma \vdash q{:}P \wedge Q}{\Gamma \vdash \pi_2(q){:}Q} \ ,$$

where $\pi_2(q)$ denotes **classic** Q $(\lambda H{:}Q \Rightarrow\perp .\ q(\lambda H'{:}P.\ H))$, which has as type Q.

$$\vee\textit{-intro}_1 \quad \frac{\Gamma \vdash P \vee Q{:}*_p \quad \Gamma \vdash p{:}P}{\Gamma \vdash \mathit{injl}\ (P \vee Q)\ p{:}P \vee Q},$$

where $\mathit{injl}\ (P \vee Q)\ p$ denotes $(\lambda H{:}P \Rightarrow\perp .\ (Hp)Q)$, which has as type the coding of $P \vee Q$.

$$\vee\textit{-intro}_2 \quad \frac{\Gamma \vdash P \vee Q{:}*_p \quad \Gamma \vdash q{:}Q}{\Gamma \vdash \mathit{injr}\ (P \vee Q)\ q{:}P \vee Q},$$

where $\mathit{injr}\ (P \vee Q)\ q$ denotes $(\lambda H{:}P \Rightarrow\perp .\ q)$, which has as type the coding of $P \vee Q$.

$$\vee\textit{-elim} \quad \frac{\Gamma \vdash p{:}P \Rightarrow R \quad \Gamma \vdash q{:}Q \Rightarrow R}{\Gamma \vdash p\nabla q{:}(P \vee Q) \Rightarrow R}$$

,where $p\nabla q$ denotes

$$(\lambda H{:}(P \Rightarrow\perp) \Rightarrow Q.\ \textbf{classic}\ R\ (\lambda H'{:}R \Rightarrow\perp .\ H'(q(H(\lambda H''{:}P.\ H'(pH'')))))),$$

which has as type the coding of $(P \vee Q) \Rightarrow R$.

$$\exists\textit{-intro} \quad \frac{\Gamma \vdash (\exists x{:}U.\ P){:}*_p \quad \Gamma \vdash p{:}P[x := t]}{\Gamma \vdash \mathit{inj}\ (\exists x{:}U.\ P)\ p\ t{:}(\exists x{:}U.\ P)},$$

where $\mathit{inj}\ (\exists x{:}U.\ P)\ p\ t$ denotes $(\lambda H{:}(\Pi x{:}U.\ P \Rightarrow\perp).\ Htp)$, which has as type the coding of $(\exists x{:}U.\ P)$.

$$\exists\textit{-elim} \quad \frac{\Gamma \vdash Q{:}*_p \quad \Gamma \vdash (\exists x{:}U.\ P){:}*_p \quad \Gamma \vdash p{:}(\forall x{:}U.\ P \Rightarrow Q)}{\Gamma \vdash \diamond\ (\exists x{:}U.\ P)\ p{:}(\exists x{:}U.\ P) \Rightarrow Q},$$

where $\diamond\ (\exists x{:}U.\ P)\ p$ denotes

$$\begin{array}{l} \lambda H{:}(\forall x{:}U.\ P \Rightarrow\perp) \Rightarrow\perp .\ \textbf{classic}\ Q \\ \qquad\qquad (\lambda H'{:}Q \Rightarrow\perp .\ H\ (\lambda x{:}U.\ (\lambda H''{:}P.\ H'(pxH'')))) \end{array},$$

which has as type the coding of $(\exists x{:}U.\ P) \Rightarrow Q$.

To present the conversion algorithm in chapter 9, we need the following two theorems:

Theorem 41 *Let $\Delta_1, A : B, \Delta_2$ be a legal context (i.e. it is possible to derive $\Delta_1, A : B, \Delta_2 \vdash *_s : \Box_s$). Then $\Delta_1, A : B, \Delta_2 \vdash A : B$.*

Theorem 42 *Let $\Delta_1, A : B, \Delta_2$ be a legal context such that Δ_1, Δ_2 is also a legal context. If $\Delta_1, \Delta_2 \vdash C : D$ then $\Delta_1, A : B, \Delta_2 \vdash C : D$.*

Theorem 41 is proved by induction on the length of the context, using *intro* and *weaken* rules. Theorem 42 is proved by induction on the derivation of $C : D$.

If we want to add Leibniz equality to $\lambda P-$, it suffices to add the following rules ($\odot$ denotes a special variable, that will never occur in any formula. It is used here as a placeholder for expressions that are replaced when using Leibniz substitution):

$$\textit{=-form} \qquad \frac{\begin{array}{c}\Gamma \vdash U{:}*_s \\ \Gamma \vdash e_1{:}U \\ \Gamma \vdash e_2{:}U\end{array}}{\Gamma \vdash e_1 = e_2{:}*_p}$$

$$\textit{reflex} \qquad \frac{\begin{array}{c}\Gamma \vdash U{:}*_s \\ \Gamma \vdash e{:}U\end{array}}{\Gamma \vdash \mathit{refl}\ e{:}e = e}$$

$$\textit{Leibniz} \qquad \frac{\begin{array}{c}\Gamma \vdash Q[\odot := e_1]{:}*_p \\ \Gamma \vdash p{:}Q[\odot := e_1] \\ \Gamma \vdash e{:}e_1 = e_2\end{array}}{\Gamma \vdash \mathit{leib}\ Q\ p\ e{:}Q[\odot := e_2]}$$

We comment on the new rules:

=-form This rule states that if expressions e_1 and e_2 both have the same set-type U, $e_1 = e_2$ is a proper formula in the logic.

reflex If e is an expression of a data-type U, then $e = e$ holds by the reflexivity of equality.

Leibniz If $Q[\odot := e_1]$ is a proper formula in the logic and this formula holds by means of a proof p, then a proof e of $e_1 = e_2$ can be used to prove $Q[\odot := e_2]$. The $\odot$ takes the place of a variable which is replaced by

e_1 and e_2 respectively. One can think of Q as a template that yields a logical formula once $\odot$ is replaced by a proper expression.

Other important rules associated with Leibniz equality, like associativity and commutativity, can be derived from the rules given above. Therefore, we will introduce them here as derived rules and their shorthand notation:

$$\textit{assoc} \quad \frac{\begin{array}{c}\Gamma \vdash p{:}e_1 = e_2 \\ \Gamma \vdash q{:}e_2 = e_3\end{array}}{\Gamma \vdash \textit{assoc}\ p\ q{:}e_1 = e_3},$$

where *assoc* p q denotes *leib* $(e_1 = \odot)\ p\ q$, which has type $e_1 = e_3$.

$$\textit{comm} \quad \frac{\Gamma \vdash p{:}e_1 = e_2}{\Gamma \vdash \textit{comm}\ p{:}e_2 = e_1},$$

where *comm* p denotes *leib* $(\odot = e_1)\ (\textit{refl}\ e_1)\ p$, which has type $e_2 = e_1$.

$$\textit{E-Leibniz} \quad \frac{\begin{array}{c}\Gamma \vdash U{:}*_s \\ \Gamma \vdash E[\odot := e_1]{:}U \\ \Gamma \vdash p{:}e_1 = e_2\end{array}}{\Gamma \vdash \textit{eleib}\ E\ p{:}E[\odot := e_1] = E[\odot := e_2]},$$

where *eleib* E p denotes *leib* $(E[\odot := e_1] = E)\ (\textit{refl}\ E[\odot := e_1])\ p$, which has type $E[\odot := e_1] = E[\odot := e_2]$.

There is also a possible extension for the natural numbers, including natural induction. The rules needed for this are (note that the natural numbers are not the only possible interpretation of the type *nat*):

$$\textit{nat-form} \qquad \langle\,\rangle \vdash \textit{nat}{:}*_s$$

$$\textit{zero-form} \qquad \langle\,\rangle \vdash 0{:}\textit{nat}$$

$$\textit{succ} \qquad \frac{\Gamma \vdash x{:}\textit{nat}}{\Gamma \vdash s(x){:}\textit{nat}}$$

$$\textit{induction} \quad \frac{\begin{array}{l}\Gamma \vdash p_0{:}Q[x := 0] \\ \Gamma \vdash p_i{:}(\forall x{:}\textit{nat}.\ Q \Rightarrow Q[x := s(x)])\end{array}}{\Gamma \vdash \textit{nat\_ind}\ p_0\ p_i{:}(\forall x{:}\textit{nat}.\ Q)}$$

We finish the discussion of $\lambda P-$ with comments on the rules for natural numbers:

nat-form States that *nat* is a set-type.

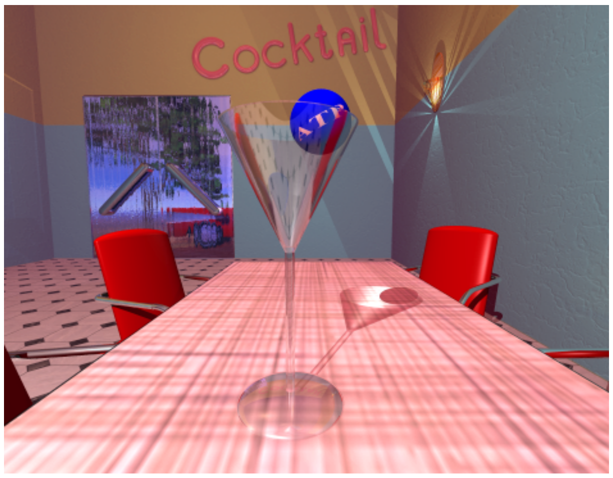
zero-form States that 0 is a natural number.

succ States that each natural number x has a successor denoted by $s(x)$.

induction If for a formula Q, possibly containing occurrences of x, it can be proved that $Q[x := 0]$ holds and also that for each x if Q holds then $Q[x := s(x)]$ holds, then Q will hold for all x. The other popular version of induction

$$\frac{\Gamma \vdash p{:}(\forall x : nat.(\forall y : nat.y < x \Rightarrow Q(y)) \Rightarrow Q(x))}{\Gamma \vdash nat\_ind'\ p{:}(\forall x : nat.Q(x))}$$

will not be used, since it requires us to define the smaller-than relation on natural numbers.



Chapter 7

Automated Theorem Proving

In this chapter, we describe two methods to automatically construct proofs. In part I we already found that the user will have to provide many simple proofs to prove correctness of her program. To increase the usability of the tool, we want to automate the construction of (nearly) trivial proofs. In the end, this will also have a positive effect on the acceptance of the tool amongst users.

7.1 Resolution

7.1.1 The Resolution Method

Resolution based theorem provers use the fact that if formula P is a tautology, then $\neg P$ does not hold in any model and vice versa. In order to prove a theorem P, a set C of formulas is created, such that there is a model of $\neg P$ if and only if there is a model of C. A model is said to be a model of C, if it is a model of all formulas in C. The set C is then expanded by adding new formulas, computed from existing formulas in C, such that the set of models of C does not change. As soon as a formula is added that cannot possibly hold in any model, we know that C cannot possibly hold in any model. Hence, $\neg P$ cannot hold in any model and therefore, P must hold in all models as was to be proved.

The set C is called a *clause set* and all formulas in C are called *clauses*. A clause A is a formula of the form $A_1 \vee \ldots \vee A_n$, where every A_i is a literal (a literal is an atomic formula or a negation of an atomic formula; definition 6 in chapter 5). Since resolution implicitly uses commutativity and associativity of $\vee$, a clause is usually considered to be a set $\{A_1, \ldots, A_n\}$ of literals. As a result, we can assume that literals do not occur more than once in a clause. Clauses can contain free variables. These free variables are implicitly universally quantified. The empty clause is denoted by $\Box$ and is invalid in all models. To avoid problems with the special literals $False$ and $\neg False$, we modify the clause set as follows:

- If $False$ occurs in a clause, it is removed from the clause.
- If $\neg False$ occurs in a clause, this clause is removed from the clause set C.

Note that the modified clause set is valid in the same set of models as the original clause set. Hence, we can assume without loss of generality that C does not contain the symbol $False$.

How the clause set is constructed will be explained later, but first we will explain how the clause set is expanded in an attempt to prove a theorem. As stated before, the resolution method derives new clauses from existing clauses that are valid in the same models and adds them to C. Hence, if the empty clause $\Box$ can be derived from existing clauses, C does not hold in any model and therefore P holds in all models.

To derive new clauses from existing clauses, resolution methods only need a single rule: the resolution rule. We will give its definition, but first we define *proper substitutions*:

Definition 43 (Proper Substitution)
A substitution θ (a partial mapping from variables to terms) is called proper*, if variables in the domain of θ do not occur in any image of θ. Formally, if $x \in Dom(\theta)$ then $\forall y \in Dom(\theta).x \notin FV(\theta(y))$. We write $A\theta$ to denote the clause or literal A in which every free variable x which is in the domain of θ is replaced by $\theta(x)$.*

Definition 44 (The Resolution rule)
Let $A = \{A_1, \ldots, A_n\}$ and $B = \{B_1, \ldots, B_m\}$ be two clauses in C, with n and m both non-zero and all A_i and B_j literals. Let θ be a proper substitution. Then, if $A_1\theta$ is syntactically equal to $\neg B_1\theta$ (i.e. θ is a unifier of A_1 and $\neg B_1$), we define the resolvent R of A and B for θ as

$$R = (A\theta \setminus \{A_1\theta\}) \cup (B\theta \setminus \{B_1\theta\})$$

This rule is sound in the sense that if M is a model of A and B, then M is a model of R. Free variables in the result R are independent of free variables in clauses A and B and may be renamed whenever needed. This is based on the fact that all free variables are implicitly universally *quantified.*

The resolution procedure now computes resolvents from the clauses in its clause set C and adds them to C. The models in which C holds do not change by this operation. If eventually the empty clause $\Box$ is derived and added to C, the procedure stops. In this case it is proved that there exist no models for C. If the procedure does not encounter $\Box$ during its computations, it cannot conclude that the theorem is incorrect: after all, it might be the case that the procedure is simply unable to *find* a derivation of $\Box$, even though it exists.

The resolution method is refutation complete, i.e. if there are no models for C, then the empty clause can be derived. For a proof see [dN95].

For instance, consider the clause set $\{\{\neg Px, \neg Py\}, \{Pv, Pw\}\}$, where P is a unary predicate and x, y, v, and w are free variables (implicitly universally quantified). Let θ be the substitution $[x \mapsto v, y \mapsto v, w \mapsto v]$. Then we can construct the following derivation:

(1)	$\{\neg Px, \neg Py\}$	given clause
(2)	$\{Pv, Pw\}$	given clause
(3)	$\square$	derived from (1) and (2) using θ

Note that it is important that clauses are treated as *sets*. The unifier θ used to derive the empty clause in the example above is found by ad-hoc inspection of the clause set. There are general methods to find unifiers: so called unification algorithms. These algorithms compare two terms (in the case above this could be $\neg Px$ and $\neg Pv$) and derive a unifying substitution if possible. However, these algorithms obtain what is called the most general unifier, which is less specific (in the example this would be either $[x \mapsto v]$ or $[v \mapsto x]$, which yield the same result upon renaming of the variables). Unfortunately, completeness is then lost. We will explain this in more detail, after providing a more formal definition of unifiers and most general unifiers.

Definition 45 (Unifier)
Let A and B be terms, with possibly free variables. Let θ be a substitution, such that $A\theta = B\theta$ and for all $x \in Dom(\theta)$ we have $x \notin FV(A\theta)$ and $x \notin FV(B\theta)$. Then θ is called a unifier of A and B.

Definition 46 (Most General Unifier)
If θ is a unifier of terms A and B, such that any unifier σ of A and B can be written as $\phi \circ \theta$ for some substitution ϕ, then θ is called a most general unifier (mgu) of A and B. Most general unifiers are unique but for renaming of the variables. Hence, a mgu is a substitution that makes two terms equal, while being as little specific as possible.

Clearly, the unifier $[x \mapsto v, y \mapsto v, w \mapsto v]$ from the example is not a mgu: We only need $[x \mapsto v]$ to unify $\neg Px$ with $\neg Pv$ and also it can be written as $[y \mapsto v, w \mapsto v] \circ [x \mapsto v]$ (note that $[x \mapsto v]$ *is* the most general unifier of $\neg Px$ and $\neg Pv$). Also the given unifier of clauses (1) and (2) is not equal to the mgu $[x \mapsto v]$ regardless of any renaming of the variables.

To unify $\neg Px$ and $\neg Pv$ from the example, the variables y and w need not to be substituted. Hence, when using the mgu for these formulas, the result will contain again two literals with two variables. In fact, every result obtained by using the mgu will have this property, regardless of which clauses and which literals are selected. Hence, the resolution algorithm using only mgu's will end up with the following clause set:

(1)	$\{\neg Px, \neg Py\}$	given clause
(2)	$\{Pv, Pw\}$	given clause
(3)	$\{\neg Py', Pw'\}$	derived from (1) and (2) using $[x \mapsto v]$

The derived clause (3) is written as $\{\neg Py', Pw'\}$ instead of $\{\neg Pv, Pw'\}$ to indicate that all variables occurring in at are implicitly universally quantified, and are independent of variables in the other clauses.

All clauses subsequently derived from this clause set already occur in it. Hence, depending upon the implementation, the program will either attempt to derive new clauses indefinitely or it will exit failing to find a proof. This does not mean that the resolution method claims that the clause set is satisfiable. It merely cannot prove otherwise. In the literature, it is often left unmentioned that this incompleteness is introduced by using most general unifiers to apply the resolution rule.

However, since efficient algorithms exist to compute most general unifiers (see [MM82]) and no information is given whether other unifiers yield better results, there is another rule often used by resolution based theorem provers: the factorization rule.

Definition 47 (The Factorization rule)
Let $A = \{A_1, \ldots, A_n\}$ be a clause and suppose that θ is a unifier of A_1 and one of the A_i with $i \in \{2, \ldots n\}$, then the factor of A with respect to θ is defined as

$$F = \{A_2, \ldots A_n\}\theta.$$

This rule is sound in the sense that M is a model of A, if and only if M is a model of F.

The factorisation rule is applied slightly different than the resolution rule: replace clause A in clause set C with clause F, where F is a factor of A with respect to a unifier θ.

A resolution based theorem prover applying resolution and factorization using only most general unifiers is again complete. The advantage is now that unifiers used in factorization can be found using the structure of the terms being unified. That the same substitution we use for x should also be used for y follows from factorization and is not given ad-hoc without argument. For instance, it is now possible to derive the empty clause from our example as follows:

(1)	$\{\neg Px, \neg Py\}$	given clause
(2)	$\{Pv, Pw\}$	given clause
(3)	$\{\neg Px\}$	derived from (1) using $[y \mapsto x]$
(4)	$\{Pv\}$	derived from (2) using $[w \mapsto v]$
(5)	$\square$	derived from (3) and (4) using $[x \mapsto v]$

Note that the composition of the substitutions that are used in this derivation yield the original substitution proposed for the first derivation.

7.1.2 Generating Clause sets

The most important and most difficult step in generating the clause set is the skolemisation of P. Skolemising P will yield a formula P' without quantifiers. Instead, P' contains free variables and new function symbols representing skolem-functions. The free variables in the skolemised formula represent arbitrary terms (i.e. they are implicitly *universally* quantified). The skolem-functions, that are always applied to a series of variables, represent terms whose existence is claimed. The intuition behind this is as follows: if the original formula claimed the existence of a term in a certain context, then it is replaced during skolemisation by a function application that is intended to represent this term, given the context. The formal definition of the skolem-form of a formula is as follows:

Definition 48 (Skolem Normal Form)
Let P be a closed formula in first order predicate logic. Then the skolem normal form of P is computed by the function skol defined recursively as given in figure 7.1.

For instance, the skolemised form of $(\forall x{:}U.\ (\exists y{:}U.\ Pxy))$ is $Px(sx)$, where s is the skolem function and sx represents the required y, given the context x.

Skolemisation has the property, that a model for a formula P exists if and only if a model for the skolemised form of P exists. Note that these models are not the same, since a model for the skolemised form also needs the

$$\begin{array}{lll}
skol(A) & = A & \text{if } A \text{ is atomic.} \\
skol(\neg A) & = \neg nskol(A) & \\
skol(A \wedge B) & = skol(A) \wedge skol(B) & \\
skol(A \vee B) & = skol(A) \vee skol(B) & \\
skol(A \Rightarrow B) & = nskol(A) \Rightarrow skol(B) & \\
skol(\forall x{:}U.\ A) & = skol(A[x := y]) & \text{where } y \text{ is a fresh variable.} \\
skol(\exists x{:}U.\ A) & = skol(A[x := s x_1 \ldots x_n]) & \text{where } s \text{ is a fresh skolem-} \\
 & & \text{function with arity } n\text{, and} \\
 & & \{x_1, \ldots, x_n\} = FV(A) \setminus \{x\}
\end{array}$$

$$\begin{array}{lll}
nskol(A) & = A & \text{if } A \text{ is atomic.} \\
nskol(\neg A) & = \neg skol(A) & \\
nskol(A \wedge B) & = nskol(A) \wedge nskol(B) & \\
nskol(A \vee B) & = nskol(A) \vee nskol(B) & \\
nskol(A \Rightarrow B) & = skol(A) \Rightarrow nskol(B) & \\
nskol(\forall x{:}U.\ A) & = nskol(A[x := s x_1 \ldots x_n]) & \text{where } s \text{ is a fresh skolem-} \\
 & & \text{function with arity } n\text{, and} \\
 & & \{x_1, \ldots, x_n\} = FV(A) \setminus \{x\} \\
nskol(\exists x{:}U.\ A) & = nskol(A[x := y]) & \text{where } y \text{ is a fresh variable.}
\end{array}$$

Figure 7.1: Skolemization of formulas.

correct interpretation of the skolem functions. For a proof of this property see [dN95].

Conjunctive Normal Form (CNF)

Once a skolemised version of formula P is available, it is rewritten into conjunctive normal form. A formula is in conjunctive normal form if it consists of a series of conjuncts, each of which consists of a series of disjuncts of literals. Hence, negations only occur in front of atomic formulas.

Definition 49 (Conjunctive Normal Form)
The conjunctive normal form of a formula is the normal form of the formula with respect to the two rewrite systems given in figure 7.2 respectively. The first system eliminates implications and pushes negations towards atomic formulas. The second rewrite system eliminates conjunctions that appear as subformulas of disjuncts.

$$\begin{array}{ll} \neg\neg A & \rightarrow A \\ A \Rightarrow B & \rightarrow \neg A \vee B \\ \neg(A \wedge B) & \rightarrow \neg A \vee \neg B \\ \neg(A \vee B) & \rightarrow \neg A \wedge \neg B \end{array}$$

$$\begin{array}{l} A \vee (B \wedge C) \rightarrow (A \vee B) \wedge (A \vee C) \\ (A \wedge B) \vee C \rightarrow (A \vee C) \wedge (B \vee C) \end{array}$$

Figure 7.2: Rewrite systems to compute the conjunctive normal form of a formula.

The clause set is obtained by using every conjunct of the conjunctive normal form as a single clause. Every conjunct exists of a disjunction of literals, which are collected in a set to form the clause. Finally, *False* symbols are eliminated from the clause set as described earlier. Note that it is possible that the empty clause $\Box$ already exists in the initial clause set. In such a case, no further action is required.

We will conclude the discussion of the resolution method with two examples. First, consider the theorem

$$((\exists x : U.\forall y : U.P(x, y)) \Rightarrow (\forall q : U.\exists p : U.P(p, q))).$$

To compute the clause set, we first skolemize the negation of the theorem, which yields

$$\neg(P(s_1(), y) \Rightarrow P(p, s_2())),$$

where s_1 and s_2 are skolem functions and y and p are implicitly universally quantified variables. Computing the conjunctive normal form of this result, we obtain

$$P(s_1(), y) \wedge \neg P(p, s_2()),$$

and hence clause set C consists of the clauses $\{P(s_1(), y)\}$ and $\{\neg P(p, s_2())\}$. The empty clause can now easily be derived using the resolution rule for these two clauses and the most general unifier $[y \mapsto s_2(), p \mapsto s_1()]$.

The second example will produce the clause set used in earlier examples. Suppose we want to use the resolution method to prove

$$A = \exists x, y, v, w{:}U.\ \neg(Px \vee Py) \vee (Pv \wedge Pw),$$

then we generate a clause set for $\neg A$. First, we compute the skolemised form of $\neg A$, which yields

$$\neg(\neg(Px' \vee Py') \vee (Pv' \wedge Pw')),$$

where x', y', v' and w' are the fresh variables substituted for x, y, v and w respectively. After applying the first rewrite system, we get

$$(Px' \vee Py') \wedge (\neg Pv' \vee \neg Pw'),$$

which is already in CNF. The clause set C corresponding to $\neg A$ then becomes $\{\{Px', Py'\}, \{\neg Pv', \neg Pw'\}\}$, for which a resolution proof has been given before. Consequently, the skolemised form of $\neg A$ has no model. So, $\neg A$ itself has no model and hence A is valid (and provable).

7.2 Tableaux

Another method to prove formulas in first order logic is the method of semantic tableaux. Like in resolution based theorem proving, the idea is that if a formula is a tautology then its negation does not hold in any model and vice versa. A tableau based theorem prover performs an exhaustive search for a model for the negation $\neg P$ of a formula P. If no model can be found for $\neg P$ then obviously $\neg P$ does not hold in any model and hence, P does hold in all models.

7.2.1 The Tableau Method

Tableau methods, opposed to resolution methods, attempt to construct a model for $\neg P$ by syntactic decomposition of the formula. Therefore, tableau methods can be used for any formula in first order logic without modification of the formula. Tableau methods are called constructive in the sense that when a model for $\neg P$ is found, one actually has a counter-example for P. Also, when a search is finished and no model for $\neg P$ is found, there is a representation of the entire proof of P. This representation is a labeled tree called the tableau. The labels are sets of formulas.

A tableau is constructed in the following manner:

1. Start with a single node, labeled with $\{\neg P\}$, where P is the formula to be proved.

2. Select a leaf from the partially constructed tree. Select from the corresponding label L a formula X to which one of the rules from figure 7.3 can be applied. Extend the leaf with a number of nodes equal to the

special rule $$\frac{\neg\neg P}{P}$$

α rule $$\frac{P \wedge Q}{P,\ Q} \qquad \frac{\neg(P \Rightarrow Q)}{P,\ \neg Q} \qquad \frac{\neg(P \vee Q)}{\neg P,\ \neg Q}$$

β rule $$\frac{\neg(P \wedge Q)}{\neg P \mid \neg Q} \qquad \frac{P \Rightarrow Q}{\neg P \mid Q} \qquad \frac{P \vee Q}{P \mid Q}$$

γ rule $$\frac{\neg\exists x : U.P}{\neg\exists x : U.P,\ \neg P[x := t]} \qquad \frac{\forall x : U.P}{\forall x : U.P,\ P[x := t]}$$

δ rule $$\frac{\exists x : U.P}{P[x := \alpha]} \qquad \frac{\neg\forall x : U.P}{\neg P[x := \alpha]}$$

Figure 7.3: Rules for the construction of tableaux for first order classical logic. α represents a fresh variable of type U. t represents an arbitrary term of type U.

number of conclusions of the rule. Conclusions are separated by a '|' and can contain several formulas separated by a ','. A successor node is labeled with $(L \setminus \{X\}) \cup Y$, where Y is the conclusion for which the successor was created. There exists a model for at least one of the successor nodes if and only if there exists a model for the parent node.

3. Repeat step 2 until:

   (a) Every leaf either contains the special predicate symbol *Falsum* or it contains both an X and $\neg X$ for some formula X. X may be different for every leaf. Such a leaf is called closed. If all leaves of a tableau are closed, the tableau itself is also called closed.

   (b) There exists a non-closed leaf which contains only literals different from *Falsum*. Such a leaf actually provides an interpretation, and hence a model, in which the original formula P does not hold, hence the tableau provides a counterexample.

The process of constructing a tableau may be non-terminating. Therefore, implementations of tableau methods usually use limits on the size of the tableau or the time used in order to guarantee termination. If the search

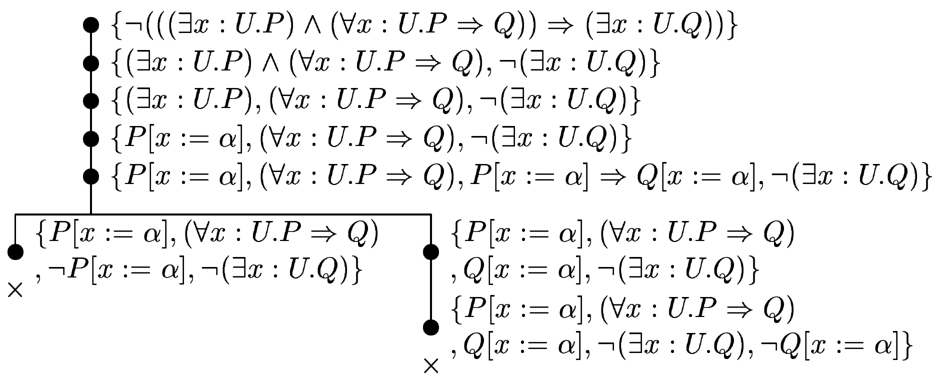

$\{\neg(((\exists x : U.P) \wedge (\forall x : U.P \Rightarrow Q)) \Rightarrow (\exists x : U.Q))\}$
$\{(\exists x : U.P) \wedge (\forall x : U.P \Rightarrow Q), \neg(\exists x : U.Q)\}$
$\{(\exists x : U.P), (\forall x : U.P \Rightarrow Q), \neg(\exists x : U.Q)\}$
$\{P[x := \alpha], (\forall x : U.P \Rightarrow Q), \neg(\exists x : U.Q)\}$
$\{P[x := \alpha], (\forall x : U.P \Rightarrow Q), P[x := \alpha] \Rightarrow Q[x := \alpha], \neg(\exists x : U.Q)\}$
$\{P[x := \alpha], (\forall x : U.P \Rightarrow Q), \neg P[x := \alpha], \neg(\exists x : U.Q)\}$
$\times$
$\{P[x := \alpha], (\forall x : U.P \Rightarrow Q), Q[x := \alpha], \neg(\exists x : U.Q)\}$
$\{P[x := \alpha], (\forall x : U.P \Rightarrow Q), Q[x := \alpha], \neg(\exists x : U.Q), \neg Q[x := \alpha]\}$
$\times$

Figure 7.4: A tableau proof for $((\exists x : U.P) \wedge (\forall x : U.P \Rightarrow Q)) \Rightarrow (\exists x : U.Q)$.

is aborted for one of these reasons, no conclusions can be drawn about the original formula. In that case, one does neither have a counterexample, nor a proof.

The method of semantic tableaux is complete in the sense that for every formula P for which $\models P$, a closed tableau exists. However, this does not imply that it can be constructed by an algorithm in a limited amount of time. Many heuristics are used in order to allow an implementation to cover a larger area of provable formulae. Discussing all these heuristics is beyond the scope of this thesis. See, for instance, [DGHP99].

A Tableau as a Proof

Figure 7.4 shows an example of a tableau that proves the formula

$$((\exists x : U.P) \wedge (\forall x : U.P \Rightarrow Q)) \Rightarrow (\exists x : U.Q).$$

The $\times$ below a leaf means that the leaf is closed.

A Tableau as a Counterexample

From the tableau in figure 7.5 we will derive a counterexample for

$$\neg(\exists x : U.(P \wedge Q) \wedge (P \Rightarrow Q)).$$

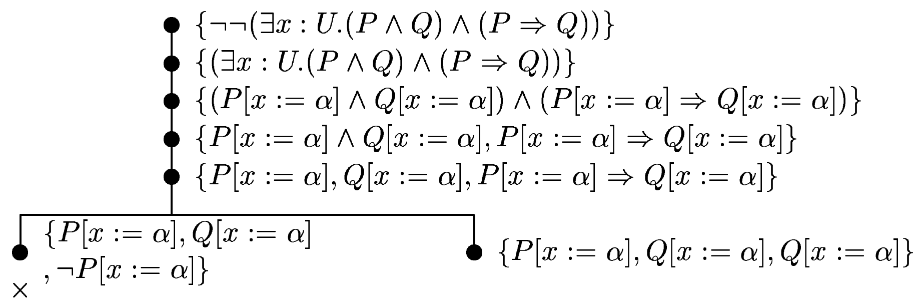

$\{\neg\neg(\exists x : U.(P \wedge Q) \wedge (P \Rightarrow Q))\}$
$\{(\exists x : U.(P \wedge Q) \wedge (P \Rightarrow Q))\}$
$\{(P[x := \alpha] \wedge Q[x := \alpha]) \wedge (P[x := \alpha] \Rightarrow Q[x := \alpha])\}$
$\{P[x := \alpha] \wedge Q[x := \alpha], P[x := \alpha] \Rightarrow Q[x := \alpha]\}$
$\{P[x := \alpha], Q[x := \alpha], P[x := \alpha] \Rightarrow Q[x := \alpha]\}$
$\{P[x := \alpha], Q[x := \alpha]$
$, \neg P[x := \alpha]\}$
$\times$
$\{P[x := \alpha], Q[x := \alpha], Q[x := \alpha]\}$

Figure 7.5: A tableau used to find a counterexample for $\neg(\exists x : U.(P \wedge Q) \wedge (P \Rightarrow Q))$.

In general, a counterexample derived from a tableau is an interpretation in which the formula we tried to prove does not hold. To derive such an interpretation from an open branch in a tableau we only have to create an interpretation in which all the formulas in the label of the open branch are true. Since the search for a closed tableau terminated with an open leaf, we know that all formulas in the label of this leaf are literals. From these literals, we can easily construct an interpretation that serves as a counter-example of the original theorem.

In our example, we derive the following interpretation from the open branch with label $P[x := \alpha], Q[x := \alpha], Q[x := \alpha]$: Take an arbitrary closed term u of type U. Choose the interpretation I such that $[P[x := u]] = \texttt{t}$ and $[Q[x := u]] = \texttt{t}$. This is possible, since by the nature of an open branch P is a predicate symbol. That the chosen interpretation is indeed a counterexample can easily be checked: In the model M^I we have $M^I((P[x := u] \wedge Q[x := u]) \wedge (P[x := u] \Rightarrow Q[x := u])) = \texttt{t}$ and hence, $M^I(\exists x : U.(P \wedge Q) \wedge (P \Rightarrow Q)) = \texttt{t}$ and thus, $M^I(\neg(\exists x : U.(P \wedge Q) \wedge (P \Rightarrow Q))) = \texttt{f}$. It follows that the theorem we were trying to prove does not hold in all models.

7.2.2 An Interpretation of the Method of Tableaux

One can consider the construction of a tableau for a formula P to be a search for an interpretation I, such that $M^I(\neg P) = \texttt{t}$ and hence, $M^I(P) = \texttt{f}$. If such an interpretation is found, we have a proof that not in all interpretations $M^I(P) = \texttt{t}$. Branching in the tableau indicates that there are two kinds of interpretations that are candidates for the search. For instance, if $\neg(A \wedge B)$

must hold for the interpretation I, then either $\neg A$ must hold for I or $\neg B$ must hold for I. This corresponds to the tableau rule

$$\frac{\neg(A \wedge B)}{\neg A \mid \neg B}$$

If a leaf is closed then the search for an interpretation failed. No interpretation can make a formula P to hold and make $\neg P$ hold at the same time. If all leaves are closed, the search for I failed altogether. We can then conclude that no interpretation I exists for which $M^I(\neg P) = \mathtt{t}$, hence $M^I(\neg P) = \mathtt{f}$ for all interpretations I and therefore $M^I(P) = \mathtt{t}$ for all interpretations I. In short, we conclude $\models P$.

The tableau method presented here is only valid for classical first-order predicate logic. Other tableau methods exist for intuitionistic logic, modal logics and even some higher-order logics. Soundness and completeness for an intuitionistic version of semantic tableaux can be proved fully intuitionistically (see [dS93]). Soundness and completeness proofs for the method presented above can be found in [Smu68].

7.2.3 A Note on closing Leaves

When applying a γ-rule, i.e. a tableau rule for $\neg\exists x{:}U.\ P$ or $\forall x{:}U.\ P$ one uses an arbitrary term t of type U. Ideally, one should predict which term t produces a closed tableau in as little more steps as possible. However, so far we have not presented any clues about which terms are 'sensible' in this sense. In fact, it is quite difficult to choose a 'sensible' term t at the moment the rule is applied (at this point, we do not yet know which formulas will cause the leafs to close, if any). In the following paragraphs, we will discuss a method to find adequate terms as proposed in [Oph92]. An implementation of this method of tableaux as used in the programming system is presented in part III.

The idea in [Oph92] is that since it is hard to decide on a 'sensible' term to be used during the application of a γ-rule, this decision should be postponed until we have more 'information'. Instead of using a term t to be substituted for x, a single, fresh variable is substituted for x, say X. Variables such as this new X, representing substitutions that have *not* yet been executed, we call *tableau variables*. Later, when searching for terms A and $\neg A$ in a leaf, a unification algorithm is used in order to (possibly) find a substitution

θ for these new variables, such that $B\theta$ syntactically equals $\neg C\theta$ for B and C occurring in the leaf. This substitution then instantiates the newly, introduced variables that are 'sensible'.

However, there are some constraints on the values that may be substituted for the new variables. For instance, it is not allowed to substitute for a new variable X introduced by a γ-rule, a variable α introduced at a *later stage* by a δ-rule. Indeed, at the time we needed a term t and used X instead, the variable α did not yet exist. Also, if we find for X a term t by unification in one leaf, then t must be substituted for X in *all* leafs. Hence, we are not allowed to create a unifier for each leaf separately, but have to create one unifier that closes all leafs at once.

To meet these constraints a context is maintained during the computation of a tableau. This context contains all variables introduced during the application of γ- and δ-rules in the order in which they are introduced. A term that should be substituted for a variable X may contain only variables α introduced by δ-rules that occur in the context before X; i.e. those variables that were available at the time X was introduced. If a leaf is closed using a substitution θ, then this substitution is applied to the entire tableau. Otherwise it would be possible to use other substitutions for the same variables in other leafs.

An example of this is shown in figure 7.6. The contexts of the leaves, in which Greek letters are used for variables introduced by δ-rules and capitals for variables introduced by γ-rules, are displayed to the right of the labels. Note that it is not possible to close the tableau one level higher than it is. The unifier $[X \mapsto \beta, Y \mapsto \alpha]$ required for this is not valid, since β occurs in the value for X, but X occurs in the context before β does. (In retrospect, one could say that the X has been introduced 'too early"; it is needless since it does not play a role in the substitution. The Z, introduced in the final label, does the job.)

●	$\{\neg((\exists x : U.\forall y : U.Pxy) \Rightarrow (\forall q : U.\exists p : U.Ppq))\}$	$<>$
●	$\{(\exists x : U.\forall y : U.Pxy), \neg(\forall q : U.\exists p : U.Ppq))\}$	$<>$
●	$\{(\forall y : U.P\alpha y), \neg(\forall q : U.\exists p : U.Ppq)\}$	$< \alpha >$
●	$\{(\forall y : U.P\alpha y), P\alpha X, \neg(\forall q : U.\exists p : U.Ppq)\}$	$< \alpha, X >$
●	$\{(\forall y : U.P\alpha y), P\alpha X, \neg(\exists p : U.Pp\beta)\}$	$< \alpha, X, \beta >$
●	$\{(\forall y : U.P\alpha y), P\alpha X, \neg(\exists p : U.Pp\beta), \neg PY\beta\}$	$< \alpha, X, \beta, Y >$
●	$\{(\forall y : U.P\alpha y), P\alpha X, P\alpha Z, \neg(\exists p : U.Pp\beta), \neg PY\beta\}$	$< \alpha, X, \beta, Y, Z >$
$\times$	using $[Y \mapsto \alpha, Z \mapsto \beta]$	

Figure 7.6: A tableau constructed using Ophelder's method

Chapter 8

Hoare Logic

After discussing the formal logic and methods for automated proof construction, we will focus on the Hoare logic used by the programmer. The Hoare logic links programs directly to their specifications and is the formal basis for correctness proofs of programs. The specifications are written as logical formulas from the theorem prover's logic. As a result, proving correctness of a program will require proofs to be constructed within the theorem prover.

8.1 The Language *While*

Imperative languages are the most common programming languages. This is probably because they can be interpreted operationally. One can easily imagine a computer assigning values to variables and proceeding with the next action. Functional languages, however, are more mathematics-based. Logical languages are easy to read, but programming in logical languages brings along specific problems (e.g. efficiency, termination).

In this chapter, we will define a simple imperative language called *While* (see also [NN92]). Also, we will define the denotational semantics of *While* and next introduce the corresponding Hoare logic. An excellent overview of theory developed for Hoare logic can be found in [Apt81].

Definition 50 (*While*)
Let Set, V and F be a set of type symbols, variables and function symbols, respectively. Assume that a special symbol **bool** *exists in Set. Let T be the set of terms as defined in definition 1 on page 34. The set H of pseudo-programs is defined by the following abstract syntax:*

$$\begin{array}{lll} H & ::= & \textbf{skip} \mid V := T \mid \textbf{if } T \textbf{ then } H \textbf{ else } H \textbf{ fi} \mid \textbf{while } T \textbf{ do } H \textbf{ od} \\ & & \mid |[\textbf{var } V := T : Set \bullet H\ ]| \mid H; H \end{array}$$

A pseudo-program S is well-formed if and only if:

1. *For every subprogram $v := e$ occurring in S, the associated types of v and e are equal.*
2. *For every subprogram* **if** g **then** S_1 **else** S_2 **fi** *and* **while** g **do** S_3 **od** *occurring in S, the associated type of g is* **bool**.
3. *For every subprogram $|[$**var** $v := e : U \bullet S_1\ ]|$ in S, the associated types of v and e are both U.*

The language While consists of all well-formed programs in H.

$PV(S)$ denotes the set of program variables of program S (i.e. the variables that are possibly altered during execution of the program, excluding locally defined variables).

Definition 51 (Program Variables)
The definition of PV is given by:

$$\begin{array}{lll}
PV(\mathbf{skip}) & = & \emptyset \\
PV(x := e) & = & \{x\} \\
PV(\mathbf{if}\ G\ \mathbf{then}\ S1\ \mathbf{else}\ S2\ \mathbf{fi}) & = & PV(S1) \cup PV(S2) \\
PV(\mathbf{while}\ G\ \mathbf{do}\ S\ \mathbf{od}) & = & PV(S) \\
PV(|[\mathbf{var}\ x := e : U \ \bullet\ S]|) & = & PV(S) \setminus \{x\} \\
PV(S1; S2) & = & PV(S1) \cup PV(S2)
\end{array}$$

To define what it means for a well-formed program to be executed, we need the concept of a state.

A state is a mapping from program variables to values, just like the mapping of variables to values in an interpretation of a logic. Thus, we consider a state to be a part of an interpretation of a logic.

Definition 52 (State)
Let $I = (\mathcal{D}, s)$ be an interpretation of the logic used to define While, as in definition 7 on page 37. Then s restricted to application on program variables is called the state.

8.2 Denotational Semantics

The denotational semantics of a program is defined as a function $D : H \to state \to \mathcal{P}(state)$, which maps every program to a state transformer (a state transformer is a function mapping a state to a set of states). The intended meaning is that for a program S and a state s, $D(S)(s)$ is the set of possible final states in which S can terminate when executed from state s. Note that for non-terminating programs, this set will be empty. Since *While* is deterministic, the set of final states will contain at most one element. We use sets of states instead of a single output state to avoid the need for a special symbol $\perp$ to indicate non-termination.

To define the changes in states that are made by the program, we assume the interpretation of the set **bool** is defined as $s(\mathbf{bool}) = \{True, False\}$. Furthermore, we use the entire function s of the interpretation $I = (\mathcal{D}, s)$ to define the denotational semantics, but only the state part will change.

Definition 53 (Denotational Semantics of *While*)
Let $I = (\mathcal{D}, s)$ be an interpretation as defined in chapter 5. Then, the denotational semantics of While is defined as follows (we use λ here to denote real functions, not λ-terms of $\lambda P-$):

$$\begin{aligned}
&D(\mathbf{skip}) = \lambda s : state.\{s\}\\
&D(x := e) = \lambda s : state.\{s[x \mapsto s(e)]\}\\
&D(|[\mathbf{var}\ x := e : U\ \bullet\ S\ ]|) = \lambda s : state.\{s'[x \mapsto s(x)] \mid\\
&\qquad\qquad\qquad\qquad\qquad\qquad s' \in D(S)(s[x \mapsto s(e)])\}\\
&D(S_1, S_2) = SEQ(D(S_1), D(S_2)),\\
&D(\mathbf{if}\ G\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}) = COND(G, D(S_1), D(S_2)),\\
&D(\mathbf{while}\ G\ \mathbf{do}\ S\ \mathbf{od}) = \mu G,\\
&\ \textit{where } G = \lambda W : state \to \mathcal{P}(state).COND(G, SEQ(D(S), W), D(\mathbf{skip})).
\end{aligned}$$

The auxiliary functions SEQ and COND are defined as:

$$\begin{aligned}
SEQ(D_1, D_2) &= \lambda s : state. \textstyle\bigcup_{s' \in D_1(s)} D_2(s')\\
COND(G, D_1, D_2)(s) &= D_1(s)\ \textit{if}\ s(G) = True,\\
&\ D_2(s)\ \textit{if}\ s(G) = False
\end{aligned}$$

and μ is the least fixed point operator.

Defining and discussing this operator is beyond the scope of this thesis, since we will use the denotational semantics only as a stepping stone to the Hoare logic. The interested reader is referred to [NN92].

An inner block, denoted as $|[\mathbf{var}\ v := e : U\ \bullet\ S\ ]|$, allows the programmer to declare a local variable v of type U, initialized with value e, which is only used within the program S. Note that the semantics of the inner block statement restores the value of the locally declared variable to its original value. If this would not be done, the state lemma given below would not hold.

We can easily lift these semantics to operate on interpretations or even models instead of states. In the following we will do so without further comment. Also, we will speak of formulas to hold in a state, meaning it holds in any interpretation consistent with the state. Note that since $\ldots := \ldots$ and $|[\ldots]|$ cannot be used for anything but variables, the interpretation of function- and predicate symbols will not be altered by any program.

Lemma 54 (State lemma) *Let S be a program and x be a variable, such that $x \notin PV(S)$. Let $I = (\mathcal{D}, s)$ be an interpretation. Then for each $s' \in D(S)(s)$ we have*

$$s(x) = s'(x).$$

Proof: Induction on the structure of S. $\Box$

Hence, programs will only alter a limited part of a state.

The definition of While as given above differs a bit from usual language definitions in the literature. Usually, expressions and types of variables are defined explicitly and not, as above, defined using expressions and types of a logic. However, by linking the programming language to the logical language we avoid problems with expressibility (any expression in the programming language must have a corresponding expression in the specification language). Also, if we use a more powerful logic, we get a more powerful language automatically. The programming language is merely used to define an order in which computations are performed and to describe how results are stored. This difference also suggests a different linking of the logic and the programming language than usual. However, when making a choice for the link between logic and language, we will provide more arguments (see chapter 10).

8.3 Hoare Triples

Hoare triples express claims about the final state of a program related to the initial state of these programs. These claims are formulated by logical formulas rather than explicit states and hence, they deal with groups of states rather than single states.

The formula describing (properties of) the initial state is called the *precondition*. The formula describing (properties of) the final state is called the *postcondition*.

Given a precondition, a postcondition and a program we can denote two kinds of Hoare triples: those for partial correctness and those for total correctness. The difference is that in total correctness termination of the program is guaranteed, while partially correct programs may not terminate.

Definition 55 (Hoare Triples for Partial Correctness)
Let L be a logic with the sets Set, V and F as used to define While. Let P and Q be formulas of this logic and let S be a While-program. Then

$$\{P\}S\{Q\}$$

is a Hoare triple expressing the partial correctness of S with respect to P and Q. This Hoare triple is valid if and only if for all states s for which $P(s)$ is valid, $Q(s')$ is valid for all s' in $D(S)(s)$.

This kind of Hoare triple should be read as: if P holds in a state s and executing S in s yields s', then Q will hold in s'. Note that it is not claimed that a suitable s' exists, i.e. it is not claimed that S terminates.

Definition 56 (Hoare Triples for Total Correctness)
Let L be a logic with the sets Set, V and F as used to define While. Let P and Q be formulas of this logic and let S be a While-program. Then

$$\{P\}S\{\Downarrow Q\}$$

is a Hoare triple expressing the total correctness of S with respect to P and Q. This Hoare triple is valid if and only if $\{P\}S\{Q\}$ is valid and $D(S)(s)$ is not empty for any s in which P holds.

This kind of Hoare triple should be read as: if P holds in a state s, then executing S in s yields a state s', such that Q will hold in s'. Note that now it *is* claimed that a suitable s' exists, i.e. it is claimed that S will terminate.

A Hoare triple is much like a formula in that it may be true or not. However, a Hoare triple should never be considered to *be* a formula. We will illustrate this by an example: Independent of the interpretation, the following Hoare triple is valid:

$$\{x = a\}x := x + 1\{x = a + 1\}.$$

But if it were regarded a formula, then

$$\forall a : U.\{x = a\}x := x + 1\{x = a + 1\}$$

would also be a formula that is valid and hence, we could derive

$$\{x = x\}x := x + 1\{x = x + 1\}$$

to be a valid Hoare triple, which it is certainly not.

The actual problem is that there are two different models involved in the correctness of a Hoare triple instead of just one. (Namely, one for the precondition and one for the postcondition).

8.3.1 Partial Correctness versus Total Correctness

In the remainder of this thesis, we will only consider Hoare triples for partial correctness. For this, there are several reasons, which are discussed below.

Simpler logic Reasoning about partial correctness requires a simpler logic than reasoning about total correctness. For repetitive constructs in the language (e.g. a while-loop, or a recursive function) one must be able, in the case of *total* correctness, to express an upper bound for the number of repetitions performed. Hence, the logic must provide a well founded set and the required ordering of this set.

Arbitrary functions For partially correct programs, we may allow arbitrary functions in the model of the logic. For total correctness it must be claimed that all functions occurring in an expression e must be computable to ensure that $x := e$ terminates. These are restrictions at a meta-level which limit the models that are usable for the logic.

Less verification conditions Since termination of repetitive constructs in the language is not proved, less conditions have to be verified by the programmer to prove correctness of the program. Since formal derivation of a program already involves many of such verification conditions, this is can be an advantage for the programmer.

Non-terminating programs Not all programs are guaranteed to terminate. For instance, a program implementing the method of tableaux presented in chapter 7 cannot be guaranteed to terminate, since there are formulas for which infinite tableaus are generated (e.g. a tableau to prove $(\exists x{:}U.\ P(x))$).

However, if a programmer wants to prove termination of her program she can still use a system like the one presented in this thesis, provided that an extension to introduce local constants is added. She then needs to specify her own well-founded set and the required ordering relation in the logic and

```
{$P$}
while $g$ do {$P \wedge g = \mathbf{true}$}
    |[const $A := e : nat$ • {$P \wedge g = \mathbf{true} \wedge A = e$}
        ...
        {$P \wedge A > e$}
    ]| {$P$}
od
{$P \wedge g = \mathbf{false}$}
```

Figure 8.1: The user can specify termination herself, provided that local constants are available in the language.

add the termination conditions to the specification of the program. She can also restrict termination proofs to those parts of the program where termination is non-trivial. For instance, termination of a loop body could be specified by the user as given in figure 8.1.

8.4 Hoare Logic

A Hoare logic is an axiomatic derivation system to prove the validity of Hoare triples. To present the Hoare logic for *While*, we need Leibniz equality. Moreover, we need to express whether a boolean expression is mapped to *True* or *False* in an interpretation. Therefore, we assume that there are two constants **true** and **false** with $s(\mathbf{true}) = \mathit{True}$ and $s(\mathbf{false}) = \mathit{False}$. The Hoare logic for *While* then consists of the following derivation rules:

Definition 57 (Hoare Logic for While)

The Hoare logic for While is defined by the derivation rules given in figure 8.2.

We briefly describe these rules:

skip Since **skip** does not change the state, the state after termination is equal to the one the execution started in. Hence, the same propositions will hold.

[skip] $$\{P\}\mathbf{skip}\{P\}$$

[assign] $$\{P[x := e]\}x := e\{P\}$$

[if] $$\frac{\{P \wedge e = \mathbf{true}\}S_1\{Q\} \quad \{P \wedge e = \mathbf{false}\}S_2\{Q\}}{\{P\}\mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\{Q\}}$$

[while] $$\frac{\{P \wedge e = \mathbf{true}\}S\{P\}}{\{P\}\mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{od}\{P \wedge e = \mathbf{false}\}}$$

[block] $$\frac{\{P \wedge x = e\}S\{Q\}}{\{P\}|[\mathbf{var}\ x := e : U\ \bullet\ S\ ]|\{Q\}} \quad \text{if } x \notin FV(P, Q)$$

[comp] $$\frac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

[cons] $$\frac{\models P' \Rightarrow P \quad \{P\}S\{Q\} \quad \models Q \Rightarrow Q'}{\{P'\}S\{Q'\}}$$

Figure 8.2: The derivation rules of the Hoare logic for *While*.

assign If P has to hold after the value of x is changed to the value of e, then $P[x := e]$ had to hold before this assignment was performed. One is often tempted to write this axiom as $\{P\}x := e\{P[x := e]\}$; but then $\{x = 0\}x := 1\{1 = 0\}$ and $\{x < 5\}x := x + 1\{x + 1 < 5\}$ would hold.

if In any state in which P holds, e evaluates either to *True* of *False*. Depending on this, S_1 or S_2 will be executed respectively. From the premises we get that if S_1 is executed in a state in which P holds and e evaluates to *True*, then Q will hold in the resulting state. Similarly, Q will hold after executing S_2 from a state in which P holds and e evaluates to *False*. Hence, regardless of the value of e, Q will hold after execution of the if-statement.

while Operationally, the body S of the while-loop is executed as long as evaluation of the guard e yields *True*. The premise of this rule states that if S is executed in a state in which P holds and $e = \mathbf{true}$, that P will still hold upon termination of S. Hence, P remains true, regardless of the number of executions of S. The while-loop ends when evaluating e yields *False*. Hence, upon termination we have both P and $e = \mathbf{false}$.

block The block-statement introduces a local variable x and initializes it with the value of e. Since $x \notin FV(P)$, it is sufficient to have P as precondition for the block-statement. The fact that $x = e$ is established by initialization of the block. The premise claims that Q holds after execution of S from a state in which $P \wedge x = e$ holds. Since $x \notin FV(Q)$, Q will also hold after execution of the entire block. The condition $x \notin FV(P, Q)$ seems more restrictive than needed for the denotational semantics. It is needed because x is set back to the value it had before executing the block-command. In case the variable is ill-named one can use the following (provable) property of programs called α-conversion:

$$|[\mathbf{var}\ x := e : U \ \bullet \ S\ ]| \equiv |[\mathbf{var}\ y := e : U \ \bullet \ S[x := y]]|$$

for any variable $y : U \notin FV(S)$.

comp The composition statement first executes S_1 and after that executes S_2. If Q holds after executing S_1 from a state in which P holds and R holds after executing S_2 from a state in which Q holds (i.e. the final state of S_1), then R holds after executing $S_1; S_2$ in a state in which P holds.

cons The logical premises claim that P holds in all states in which P' holds and that Q' holds in all states in which Q holds. Since Q holds after executing S from a state in which P holds, Q' will also hold after executing S from a state in which P holds. Hence, Q' will also hold after executing S from a state in which P' holds. This rule is known as 'the rule of consequence'.

A proof of the soundness of these rules can be found in [NN92].

8.5 Extensions

In this section we discuss two important extensions of the language *While* introduced in this chapter. The first extension, arrays, allows new types to be constructed by the user. The second extension, procedures, allows the user to introduce (parametric) macros, which can be used in the actual program.

We will discuss each extension in a separate subsection.

8.5.1 Arrays

Many interesting problems require arrays to allow efficient solutions (for instance, the knapsack problem). In $\lambda P-$, array types are not directly supported. Using Pascal-like arrays is troublesome, since it requires the bounds of the array to be defined. Rules to extend a logic with Pascal-like arrays would therefore require rules to define the complete ordering of the natural numbers. Instead, we will discuss unbounded arrays. These can be considered to be partial functions from the natural numbers to any datatype (including arrays). In contrast to ordinary functions, partial functions represented by arrays can be altered by the program.

We can think of arrays as lists of pairs. The first element of the pair represents the domain value and the second element represents the value of the array for this domain value. When the list is extended with a new pair (i, v), the array will return value v at domain value i, overruling the old array-value for i. The value of an array in given domain value i is computed by searching the last pair in the list whose first element equals i. The second element of this pair then represents the corresponding array-value.

For instance, a user can define an array of natural numbers as follows:

$$\begin{aligned}
&array\_nat : *_s,\\
&mod(x : array\_nat, i : nat, v : nat) : array\_nat,\\
&val(x : array\_nat, i : nat) : nat,\\
&red1 : \forall x : array\_nat.\forall i : nat.\forall v : nat.\\
&\qquad\qquad\qquad val(mod(x,i,v),i) = v,\\
&red2 : \forall x : array\_nat.\forall i : nat.\forall v : nat.\forall j : nat.\\
&\qquad\qquad\qquad \neg(i = j) \Rightarrow val(mod(x,i,v),j) = val(x,j),\\
&a : array\_nat
\end{aligned}$$

Then a program can use variable a of type $array\_nat$ as an array of natural numbers. Initially, nothing is known about this array (i.e. it has an empty domain). To alter the array such that it maps domain value $index$ to value y, the assignment $a := mod(a, index, y)$ is used. To obtain the value of a at domain value $index$, the expression $val(a, index)$ is used. For instance, the array $mod(a, index, y)$ yields value y at domain value $index$, which is stated by axiom $red1$. Hence, the value of the original array a at domain value $index$ is irrelevant to this result: the value last assigned to the array at domain value $index$ determines the result. Axiom $red2$ states that the value of array $mod(a, index, y)$ at any domain value other than $index$, is just the value of array a at this domain value.

In order for arrays to be supported more directly by a logic, we assume the existence of a type symbol nat, such that in every interpretation $I = (\mathcal{D}, s)$ under consideration $s(nat) = \mathbb{N}$. In this logic, the type and term-structure is then extended as follows:

- If U is a set symbol, then $array(U)$ is also a set symbol. (The set-symbols are closed under the array operator).

- If x, i and v are terms with associated types $array(U)$, nat and U respectively, then
  $x[i/v]$ is a term with associated type $array(U)$ and
  $x(i)$ is a term with associated type U.

  Intuitively, the array $x[i/v]$ is a modified version of array x, that maps the domain value of expression i to the value of expression v. $x(i)$ is the value of array x at the domain value of expression i.

- Let $I = (\mathcal{D}, s)$ be an interpretation and let x, i, j and v be terms of type $array(U)$, nat, nat and U respectively, then
  $s(array(U)) = \mathbb{N} \hookrightarrow U$
  $s(x[i/v]) = s(x)[s(i) \mapsto s(v)]$, where
  $s(x)[s(i) \mapsto s(v)](p) = s(x)(p)$ if $p \neq s(i)$
  $s(x)[s(i) \mapsto s(v)](p) = s(v)$ if $p = s(i)$
  $s(x(i)) = s(x)(s(i))$
  Hence, the value of an array is a partial function f, possibly a modified function $f'[i \mapsto v]$ and the value of an array at a certain domain index is the value of this function at the same index.

Arrays are considered to be functions from the natural numbers to other types. The difference between an array and a function is that a modified version of a function cannot be expressed directly in the logic. A modified version of an array x can be expressed as $x[i/v]$. Hence, users of the logic can change functions in certain points (which is exactly what is done in programs using arrays).

$\lambda P-$ can be extended to match with this extended logic by adding the derivation rules given in figure 8.3. Note that these rules can be replaced by higher-order definitions and axioms in a more powerful PTS:

As usual, we comment briefly on each rule:

$$\textit{array-form}\quad \frac{\Gamma \vdash U{:}*_s}{\Gamma \vdash \textbf{array of } U{:}*_s}$$

$$\textit{array-val}\quad \frac{\begin{array}{l}\Gamma \vdash x{:}\textbf{array of } U\\ \Gamma \vdash i{:}nat\end{array}}{\Gamma \vdash x(i){:}U}$$

$$\textit{array-mod}\quad \frac{\begin{array}{l}\Gamma \vdash x{:}\textbf{array of } U\\ \Gamma \vdash i{:}nat\\ \Gamma \vdash e{:}U\end{array}}{\Gamma \vdash x[i/e]{:}\textbf{array of } U}$$

$$\textit{array-red1}\quad \frac{\Gamma \vdash x[i/e]{:}\textbf{array of } U}{\Gamma \vdash red1\ x[i/e]{:}x[i/e](i) = e}$$

$$\textit{array-red2}\quad \frac{\begin{array}{l}\Gamma \vdash x[i/e]{:}\textbf{array of } U\\ \Gamma \vdash p{:}\neg(i = j)\end{array}}{\Gamma \vdash red2\ x[i/e]\ p{:}x[i/e](j) = x(j)}$$

$$\textit{array-start}\quad \frac{\Gamma \vdash \textbf{array of } U{:}*_s}{\Gamma \vdash \delta\ U{:}\textbf{array of } U}$$

Figure 8.3: Derivation rules for array-types in $\lambda P-$

array-form This rule introduces the possibility to construct array-types, based on a data-type U.

array-val Informally, this rule claims that values of an array with type **array of** U are of type U as one would expect.

array-mod The array-modification rule allows the user to construct modified arrays based on existing arrays. I.e., if x is of type **array of** U then the same array, modified at index i, where its value should be e (i.e. $x[i/e]$) is also a valid array of type **array of** U.

array-red1 Once a theorem contains modified arrays, additional rules are needed to decompose them. The first 'reduction' rule states that the value of array $x[i/e]$ at index i is e.

array-red2 The second 'reduction' rule states that the value of $x[i/e]$ at index j is equal to the value of the original x at index j for all j different from i.

array-start This rule is given for practical reasons. Without this rule, programs will not be able to declare local variables of array-types if no expressions of such type are derivable from the context. The reason for this is that the initial value required by the *Block*-rule cannot be given. $\delta\ U$ represents an array of type **array of** U whose values are unknown. By repetitive altering values at given indices, arrays are constructed that give more information.

Note that the rules above are generalizations of the rules used to model arrays of natural numbers, given at the beginning of this subsection.

8.5.2 Procedures

Usually, a programmer does not write a program all at once. Larger programs are split into separate tasks which should be accomplished. For each task a separate sub-program is written and the 'main' program solves the entire computation problem by using the sub-programs. Often, many of the smaller tasks are equal, such that one sub-program can be used more than once, which increases the efficiency of program construction and decreases the chance of errors.

This modular approach to writing programs is utilized by the procedure mechanism. In its simplest form, the procedure mechanism allows the programmer to introduce a shorthand name for a series of statements. This name may then be used to replace exactly this series of statements. Informally, this could be denoted as in the following example:

$$\begin{array}{l} |[\textbf{proc}\ max = \textbf{if}\ x < y\ \textbf{then}\ x := y\ \textbf{else skip fi} \bullet \\ \qquad x := X; y := Y; max; y := Z; max \\ ]| \end{array}$$

where X, Y and Z are constant expressions. By definition, this program is equal to:

$$\begin{array}{l} x := X; y := Y; \textbf{if}\ x < y\ \textbf{then}\ x := y\ \textbf{else skip fi}; \\ \qquad y := Z; \textbf{if}\ x < y\ \textbf{then}\ x := y\ \textbf{else skip fi} \end{array}$$

The procedure *max* assigns the maximum of x and y to x. The program segment **if** $x < y$ **then** $x := y$ **else skip fi** in the first program is called the *body* of the procedure *max* and the last two occurrences of *max* in the first program are called the procedure calls of max.

Unfortunately, this procedure mechanism restricts the use of max to compute the maximum of variables x and y and cannot be used for variables a and b. Therefore, procedures with parameters are introduced. In a procedure with parameters, names for variables and expressions are introduced during the definition of the procedure and are replaced by real variables and expressions when the procedure-definition is used. For example:

```
|[proc $max(\textbf{var}\ a, b : nat) = \textbf{if}\ a < b\ \textbf{then}\ a := b\ \textbf{else skip fi}\ \bullet$
        $x := X; y := Y; z := Z; max(x, y); max(x, z)$
]|
```

which, by definition is equal to:

```
$x := X;\ \ y := Y;\ \ z := Z;\textbf{if}\ x < y\ \textbf{then}\ x := y\ \textbf{else skip fi};$
                               $\textbf{if}\ x < z\ \textbf{then}\ x := z\ \textbf{else skip fi}$
```

Hence, each procedure call is replaced by the procedure's body with appropriate substitutions.

The difference with the previous program is that in the last program the two uses of *max* have different meanings, simply because other parameters were used. As a consequence, the value of x will become the maximum of the three values assigned to x, y and z.

The examples above are constructed to avoid any complications. Yet, as simple as the procedure mechanism seems, there are many pitfalls. We will demonstrate this by a number of examples:

Late Binding versus Early Binding

Consider the following program with a parameterless procedure:

```
|[proc $max = \textbf{if}\ x < y\ \textbf{then}\ x := y\ \textbf{else skip fi}\ \bullet$
    |[var $x := X : nat\ \bullet\ y := Y; max$ ]|
]|
```

When renaming the local variable x to z, we get

```
|[proc $max = \textbf{if}\ x < y\ \textbf{then}\ x := y\ \textbf{else skip fi}\ \bullet$
    |[var $z := X : nat\ \bullet\ y := Y; max$ ]|
]|
```

which we would want to be equal to the first program, since intuitively the name of a local variable should not matter. But this is not the case. The first program is, by definition, equivalent with

$$|[\textbf{var } x := X : nat \ \bullet \ y := Y; \textbf{if } x < y \textbf{ then } x := y \textbf{ else skip fi }]|,$$

which effectively is equivalent with $y := Y$, since the value of x is restored to its original value at the end of the inner block. The second program is, by definition, equivalent with

$$|[\textbf{var } z := X : nat \ \bullet \ y := Y; \textbf{if } x < y \textbf{ then } x := y \textbf{ else skip fi }]|,$$

which also assigns the maximum of x and y to x.

The core of the problem is that we must define whether the variable x in the procedure body refers to the 'global' variable x defined outside the block (as in the last example) or the 'local' variable x defined within the inner block (as in the previous example).

The first option is referred to as early binding, since the variables x and y in the procedure body are bound at the time the procedure is defined. The second option is referred to as late binding, since the variables x and y become bound when the procedure is called which happens *after* it was defined. The second option is more powerful, since the procedure can be used for other variables than the 'global' variable x. On the other hand, renaming of local variables can affect the semantics of the program. For more information about early and late binding see [NN92].

Note on global variables in procedure bodies

Consider the following program containing a procedure with parameter:

```
|[var $x := X : nat$ • |[proc $p$(var $y : nat$) = $y := x \ div \ 2$;
                                              if $y < x$ then $y := 0$;
                                              else skip
                                              fi •
                              $p(a); p(x)$
                       ]|
]|
```

Looking at the procedure body, one would expect that $p(z)$ would set the value of variable z to zero, which is the case for the first call, in which a global variable a is used. However, if p is called with argument x (as is also done in the example) the variable is only divided by 2. The main reason for this is that in the latter case both x and y represent the same variable during execution of the procedure body[1]. Clearly, for the denotational semantics this behavior is not a problem, but it is not very intuitive for the programmer what the procedure does. This is captured by the fact that the axiomatic semantics, which state the meaning of a procedure more directly, cannot claim that the postcondition of $p(z)$ implies $z = 0$ for each z.

By using a syntactic restriction that no global variables may occur in the procedure body, we avoid that the postcondition of each procedure call must deal with the special case of multiple names for variables. This restriction can be checked automatically by the tool and does not have to be proved by the programmer.

Definition 58 (Procedures with Parameters)
Given a program S, variables x_1 till x_n with associated types U_1 till U_n respectively, variables y_1 till y_m with associated type V_1 till V_m and an unused identifier p, we define a procedure with parameters as follows:

$$\textbf{proc}\ p(\textbf{var}\ x_1 : U_1, \ldots, \textbf{var}\ x_n : U_n; y_1 : V_1, \ldots, y_m : V_m) = S,$$

where all x_i are called var-parameters (whose value can be altered by the procedure) and all y_i are called value-parameters (of which the procedure may only use the value, but not alter the value).

After a procedure p is defined as indicated above, it can be called as follows:

$$p(v_1, \ldots, v_n, e_1, \ldots, e_m),$$

where each v_i is a variable with associated type U_i and every e_j is an expression yielding a value with associated type V_j.

These definitions are only valid if the following conditions hold:

i *All x_i and y_i are different and $FV(S) \subseteq \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ and y_j is not altered by S in any way (i.e. it does not occur at the left hand side of an assignment statement or as argument for a var-parameter in procedure calls). Effectively, these conditions ensure that no global*

[1]Having multiple names for a single variable is called aliasing, which often occurs when using pointers. It is aliasing which makes pointer semantics extremely hard and cumbersome to work with.

variables are used in the procedure body S and that value parameters are not altered.

ii $\{v_1, \ldots, v_n\} \cap FV(e_1, \ldots, e_m) = \emptyset$ *and* $(\{v_1, \ldots, v_n\} \cup FV(e_1, \ldots, e_m)) \cap \{x_1, \ldots, x_n, y_1, \ldots, y_m\} = \emptyset$. *Effectively, these conditions ensure that every variable exists only under a single name and that altering a var-parameter does not alter the value of any of the value-parameters.*

In that case, the denotational semantics of the procedure call are defined as:

$$\begin{array}{l} D(p(v_1, \ldots, v_n, e_1, \ldots, e_m)) = \\ \quad D(|[\mathbf{var}\ y_m := e_m : V_m \bullet\ |[\ \ldots\ \bullet\ |[\mathbf{var}\ y_1 := e_1 : V_1 \bullet \\ \qquad |[\mathbf{var}\ x_n := v_n : U_n \bullet\ |[\ \ldots\ \bullet\ |[\mathbf{var}\ x_1 := v_1 : U_1 \bullet \\ \qquad\qquad S; \\ \qquad\qquad v_1 := x_1; \ldots; v_n := x_n \\ \qquad]|\ \ldots\ ]|) \end{array}$$

Hence, instead of defining the semantics directly, we use the translation of the procedure call.

Note that since we have α-conversion for block-constructs, we also have α-conversion for the procedure parameters. Hence, we can relax condition ii for procedure calls: we do not have to claim that variables used as parameters for the procedure differ from variables used in the procedure calls, since they can be renamed when necessary. In the following, we will add this flexibility to the already used variable convention (variables used as procedure parameters are considered to be bound variables).

Given the syntactical conditions i and ii, the denotational semantics of a procedure call can be understood as follows:

- Copy value-parameters $e_m \ldots e_1$ to local variables $y_m \ldots y_1$ (to avoid substitution in the actual procedure body).
- Copy var-parameter values $v_n \ldots v_1$ to local variables $x_n \ldots x_1$ (again, to avoid substitution in the actual procedure body).
- Execute the procedure body to perform the computation with the given values stored in $y_1 \ldots y_n$ and $x_1 \ldots x_n$.
- Copy the results in the var-parameters $x_1 \ldots x_n$ back to the variables $v_1 \ldots v_n$ given by the calling program.

The order of the variable-indices above are reversed only for convenience in the forthcoming discussion about the Hoare rules for procedures.

Hoare rules for procedures

In Hoare logic, we want to prove that for a given procedure named p $\{P'\}p(v_1, \ldots, v_n, e_1, \ldots, e_m)\{Q'\}$ holds. Certainly, it is sufficient to prove that $\{P'\}S'\{Q'\}$ holds, where S' is the equivalent of the procedure call $p(v_1, \ldots, v_n, e_1, \ldots, e_m)$ as given by definition 58. Using the definition of a procedure call this way is not desirable though, since for every procedure call to p we have to prove correctness of the entire body with respect to the specification.

Instead, we would like to prove $\{P\}S\{Q\}$ for the procedure body S once and then use a single rule to conclude $\{P'\}p(v_1, \ldots, v_n, e_1, \ldots, e_m)\{Q'\}$ for P' and Q' related to P and Q respectively. A procedure would then be defined along with its specification, like

$$\mathbf{proc}\ p(\mathbf{var} x_1 : U_1, \ldots, \mathbf{var} x_n : U_n, y_1 : V_1, \ldots, y_m : V_m) = \{P\}S\{Q\}.$$

Remains to find correct pre- and postcondition of a procedure call

$$p(v_1, \ldots, v_n, e_1, \ldots, e_m).$$

The most intuitive approach is to use as precondition

$$P' = P[x_i := v_i]_{i=1}^{n}[y_i := e_i]_{i=1}^{m},$$

since this is just the precondition of S with all procedure parameters replaced by their counterparts used in the procedure call. Assuming conditions i and ii of definition 58 hold, we can use the denotational definition of $p(v_1, \ldots, v_n, e_1, \ldots, e_m)$ and the block-rule to show that S is executed from a state in which

$$P' \wedge y_m = e_m \wedge \ldots \wedge y_1 = e_1 \wedge x_n = v_n \wedge \ldots \wedge x_1 = v_1$$

holds. From this it follows immediately (using Leibniz) that P holds indeed.

For the postcondition Q' we use $Q[x_i := v_i]_{i=1}^{n}[y_i := e_i]_{i=1}^{m}$, since this is just the postcondition of S with all procedure parameters replaced by their counterparts used in the procedure call. From the assignment rule we get that, upon termination of S,

$$Q[x_i := v_i]_{i=1}^{n}[y_i := e_i]_{i=1}^{m}[v_i := x_i]_{i=n}^{1}$$

must hold. Since all variables are different and, by condition ii, none of the e_i contain any x_i, this can also be written as:

$$Q[x_i := v_i]_{i=1}^{n}[v_i := x_i]_{i=n}^{i}[y := e_i]_{i=1}^{m},$$

which is equal to $Q[y_i := e_i]_{i=1}^{m}$.

S is executed from a state in which $y_m = e_m \wedge \ldots \wedge y_1 = e_1$ holds. By conditions i and ii, we know that neither the values of y_i nor the values of e_j are changed by S, hence these equalities will also hold upon termination of S. Together with Q these equations imply $Q[y_i := e_i]_{i=1}^{m}$ as requested.

Hence, the denotational unfolding of $p(v_1, \ldots, v_n, e_1, \ldots, e_m)$ is correctly annotated as follows:

```
{$P[x_i := v_i]_{i=1}^{n}[y_i := e_i]_{i=1}^{m}$}
|[**var** $y_m := e_m : V_m$ • |[ ... • |[**var** $y_1 := e_1 : V_1$ •
|[**var** $x_n := v_n : U_n$ • |[ ... • |[**var** $x_1 := v_1 : U_1$ •
        {$P[x_i := v_i]_{i=1}^{n}[y_i := e_i]_{i=1}^{m} \wedge x_1 = v_1 \wedge \ldots \wedge x_n = v_n$
                                        $\wedge y_1 = e_1 \wedge \ldots \wedge y_m = e_m$}
        {$P \wedge y_1 = e_1 \wedge \ldots \wedge y_m = e_m$}
        $S$;
        {$Q \wedge y_1 = e_1 \wedge \ldots \wedge y_m = e_m$}
        {$Q[y_i := e_i]_{i=1}^{m}$}
        $v_1 := x_1; \ldots ; v_n := x_n$
]| ... ]|
{$Q[x_i := v_i]_{i=1}^{n}[y_i := e_i]_{i=1}^{m}$}
```

Hence, we conclude that

$$\{P[x_i := v_i]_{i=1}^{n}[y_i := e_i]_{i=1}^{m}\}$$
$$p(v_1, \ldots, v_n, e_1, \ldots, e_m)$$
$$\{Q[x_i := v_i]_{i=1}^{n}[y_i := e_i]_{i=1}^{m}\}$$

holds. Therefore, a valid Hoare rule for procedure calls is:

$$[\text{call}] \quad \frac{\begin{array}{l}\textbf{proc}\ p(\textbf{var}\ x_1 : U_1, \ldots, \textbf{var}\ x_n : U_n, y_1 : V_1, \ldots, y_m : V_m) \\ \qquad = \{P\}S\{Q\}\end{array}}{\begin{array}{c}\{P[x_i := v_i]_{i=1}^{n}[y_i := e_i]_{i=1}^{m}\} \\ p(v_1, \ldots, v_n, e_1, \ldots, e_m) \\ \{Q[x_i := v_i]_{i=1}^{n}[y_i := e_i]_{i=1}^{m}\}\end{array}}$$

However, this rule is not complete, since it restricts the use of the procedure to situations in which the pre- and postcondition of the procedure call match with the pre- and postcondition of the procedure body. If we want to use procedures to solve separate parts of the programming problem, we want to use a procedure call for parts of the specification.

For instance, let *fac* be a procedure computing the factorial defined as

$$\textbf{proc}\ \mathit{fac}(\textbf{var}\ x : \mathit{nat}, n : \mathit{nat}) = \{\mathit{True}\} \ldots \{x = n!\}.$$

Now we want to write a program to compute two factorials, being specified by precondition $True$ and postcondition $x = X! \wedge y = Y!$. Then we must first rewrite the postcondition into either $x = X!$ or $y = Y!$ in order to use the procedure call $\mathit{fac}(x, X)$ or $\mathit{fac}(y, Y)$ respectively. Which version is chosen is irrelevant to the fact that half of the specification is thrown away. Hence, even though the Hoare triple $\{\mathit{True}\}\mathit{fac}(x, X); \mathit{fac}(y, Y)\{x = X! \wedge y = Y!\}$ is valid, it cannot be derived by our Hoare logic.

Therefore, let T be a formula, such that $FV(T) \cap \{v_1, \ldots, v_n\} = \emptyset$, implying that its validity will not change due to a procedure call. Without loss of generality, we can also assume that T does not contain any of the x_i and y_i used by the procedure. It is now easy to see that $\{T\}S\{T\}$ and $\{T\}v_1 := x_1; \ldots; v_n := x_n\{T\}$ hold. Hence, $\{T\}p(v_1, \ldots, v_n, e_1, \ldots, e_m)\{T\}$ also holds (the validity of T is unaffected by the procedure call) and therefore, we can introduce the following rule:

$$[\text{call\_extend}] \quad \frac{\begin{array}{c}FV(T) \cap \{v_1, \ldots, v_n\} = \emptyset \\ \{P\}p(v_1, \ldots, v_n, e_1, \ldots, e_n)\{Q\}\end{array}}{\{T \wedge P\}p(v_1, \ldots, v_n, e_1, \ldots, e_n)\{T \wedge Q\}} \ ,$$

This rule re-established completeness. Applying it to the original postcondition of the factorial example yields the pre-condition $x = \mathit{fac}(x, X) \wedge True$. This is equivalent with $x = \mathit{fac}(x, X)$. Applying the [call]-rule to this postcondition yields the precondition $True$, which was given by the original program specification.

This approach is also used in [Gor75]. Note, that even though we now have two rules to derive correctness of a procedure call, it will always be decidable which one was used by comparing the structure of the pre- and postcondition of the procedure call to those of the procedure definition in the context.

The Hoare rule to introduce a procedure is

$$[\text{proc}] \quad \frac{\{P\}S\{Q\}}{\begin{array}{l}\textbf{proc}\ p(\textbf{var}\ x_1 : U_1, \ldots, \textbf{var}\ x_n : U_n, y_1 : V_m, \ldots, y_m : V_m) \\ \quad = \{P\}S\{Q\}\end{array}} \quad ,$$

provided $FV(P) \cup FV(Q) \subseteq \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ and no y_i is altered by S (i.e. $y_i \notin PV(S)$). Again, these conditions can be checked mechanically.

These three rules allow the programmer to derive valid procedures and to use them. Formally, the entire procedure has to be derived for each procedure call made. However, the idea is that a tool maintains a library of correctly derived procedures and allows the programmer to use these directly. This maintaining of a context can be made more explicit, but this is postponed to chapter 10.

It is also possible to define procedures more generally, allowing global variables, recursion or even mutual recursion (procedure A calling procedure B calling procedure A etc.) However, giving Hoare rules for these extended versions is a difficult and error-prone task. For instance, in [GL80] some rules were proposed that were later found unsound. In [HM96] a Hoare rule for mutually recursive procedures allowing global variables is proved sound within the HOL theorem prover.

Chapter 9

Combining PTSs and Tableaux

In this chapter we will describe an algorithm to convert closed tableaux into λ-terms of $\lambda P-$. In turn, these λ-terms can easily be transformed into λ-terms of other PTSs, provided that these other PTSs are powerful enough. The closed tableau may be produced by any tableau-based theorem prover. This gives us the capability to use existing theorem provers as a module in an implementation of $\lambda P-$ and thereby adding powerful automated theorem proving to an interactive proof system, without the danger of extending our logic in an unforeseen way. If there is enough trust in the correctness of the implementation of the automatic theorem prover we can also use a special token to encode that the proof can be constructed using the ATP. We then do not have to actually convert the tableau and store the large λ-term that is the result of converting the tableau. The ATP can then reconstruct the tableau and convert it into a λ-term on request; for instance, if we want to communicate our proof to somebody using a different theorem prover based on λ-calculus. The translation of tableaux into λ-terms is also described in [Fra00].

The conversion is done in a structured way: for similar rules of the tableau method, similar conversion steps are performed. The classes of similar rules of the tableau method are usually called α-, β-, γ- and δ-rules. In figure 7.3 on page 76 each class is depicted in one row. The top row displays the special rule, which is very simple to convert. In figure 9.1 for each class the structure of the rules is depicted. Our conversion algorithm will have one case for every class of rules.

$$\textit{special} \quad \frac{\neg\neg P}{P}$$

$$\alpha \quad \frac{E(P,Q)}{E_1(P),\ E_2(Q)} \qquad \beta \quad \frac{E(P,Q)}{E_1(P) \mid E_2(Q)}$$

$$\gamma \quad \frac{E(U,P)}{E(U,P), E'(P)^x_t} \qquad \delta \quad \frac{E(U,P)}{E'(P)^x_\theta}$$

t a term of type U $\quad$ θ new variable of type U

Figure 9.1: Structure of the different classes of tableau rules.

A full conversion of a tableau consists of two steps: First one has to construct a PTS-context for each node in the tableau. These contexts are constructed from top to bottom. The tableau labels are related to the contexts in the sense that every formula in the tableau label is the type of a variable in the corresponding PTS-context. The contexts, however, also contain variables. To modify contexts of $\lambda P-$ we will intensively use theorem 41 and theorem 42 on page 64. Second, a contradiction, valid in the context of the root node of the tableau, is constructed. This contradiction is constructed from bottom to top. During this second construction it is important that the PTS-contexts of child nodes can be used to derive contradictions in parent nodes.

The interdependence of these two steps in best understood if we present both constructions interleaved. We present the conversion algorithm top to bottom: we always start describing how the contexts for child nodes are derived from the context of the parent node. Next, we describe how a contradiction is derived in the context of the parent node from contradictions derived in the contexts of the child nodes. However, we start showing how the actual proof of the theorem is constructed from the contradiction derived in the context of the root node.

9.1 Converting the Initial Tableau

The tableau starts with a node labeled by $\neg P$ and the initial context for $\lambda P-$ will be $\Gamma_{\mathcal{L}}, p : \neg P$. As explained in section 7.2 the tableau represents a contradiction derived from $\neg P$ and hence, converting the tableau should result in a contradiction $c :\perp$ derived from the context $\Gamma_{\mathcal{L}}, p : \neg P$. The validity of P in $\lambda P-$ is then given by $\Gamma_{\mathcal{L}} \vdash \mathit{classic}\ P\ (\lambda p : \neg P.c) : P$.

9.2 Converting Applications of Tableau Rules

To derive a contradiction valid in the context of the current node, we first construct a context for the successor-nodes. By recursion, we get a contradiction valid in that context. From this contradiction, we construct the contradiction valid in the context of the current node. Suppose the context corresponding to the current node is denoted by $\Gamma_{\mathcal{L}}, \Delta_1, x : X, \Delta_2$, where X is the proposition to which the tableau rule was applied. The context of the successor-node(s) will be stated for each case separately. For each type of node we will describe the construction of the contradiction.

9.2.1 Conversion for the Special Rule

Our first case will deal with the special tableau-rule:

$$\frac{\neg\neg P}{P}$$

We have to derive a contradiction c from a node with context $\Gamma_{\mathcal{L}}, \Delta_1, o : \neg\neg P, \Delta_2$. First, we create for the successor-node the corresponding context

$\Gamma_\mathcal{L}, \Delta_1, \Delta_2, p : P$. The assumption $p : P$ is put at the end of the context in order to ease the derivation given below. Then, by recursion, we derive a contradiction c from this successor node, hence we have $\Gamma_\mathcal{L}, \Delta_1, \Delta_2, p : P \vdash c :\perp$. Then the contradiction we seek is derived as shown in derivation 9.1.

$$\begin{array}{ll}
 & \{ \text{ induction hypothesis } \} \\
(0) & \Gamma_\mathcal{L}, \Delta_1, \Delta_2, p : P \vdash c :\perp \\
 & \{ \ \Pi\text{-}\mathit{intro} \text{ on } (0) \ \} \\
(1) & \Gamma_\mathcal{L}, \Delta_1, \Delta_2 \vdash (\lambda p : P.c) : P \Rightarrow\perp \\
 & \{ \text{ see remark 9.2.1 and theorem 41 } \} \\
(2) & \Gamma_\mathcal{L}, \Delta_1, o : (P \Rightarrow\perp) \Rightarrow\perp, \Delta_2 \vdash o : (P \Rightarrow\perp) \Rightarrow\perp \\
 & \{ \text{ theorem 42 on (1) } \} \\
(3) & \Gamma_\mathcal{L}, \Delta_1, o : (P \Rightarrow\perp) \Rightarrow\perp, \Delta_2 \vdash (\lambda p : P.c) : P \Rightarrow\perp \\
 & \{ \ \Pi\text{-}\mathit{elim} \text{ on (2) and (3) } \} \\
(4) & \Gamma_\mathcal{L}, \Delta_1, o : (P \Rightarrow\perp) \Rightarrow\perp, \Delta_2 \vdash o \ (\lambda p : P.c) :\perp
\end{array}$$

Derivation 9.1: Derivation structure for the Special Rule.

Remark 9.2.1 *Formally we also need to derive types in order to apply the PTS-rules. For instance in step* (2), *we indirectly apply intro on the type* $(P \Rightarrow\perp) \Rightarrow\perp$ *by using theorem 41, but for this we also need a type judgment saying* $\Gamma_\mathcal{L}, \Delta_1 \vdash (P \Rightarrow\perp) \Rightarrow\perp: *_p$. *Such a type judgment can be derived by:*

(a)	$\Gamma_\mathcal{L}, \Delta_1 \vdash P : *_p$	*Theorem 39 and* $P \in \mathcal{L}$
(b)	$\Gamma_\mathcal{L}, \Delta_1, p : P \vdash \perp: *_p$	*Axiom of* $\lambda P-$ *and repeated weaken*
(c)	$\Gamma_\mathcal{L}, \Delta_1 \vdash P \Rightarrow\perp: *_p$	Π-form *on (a) and (b)*
(d)	$\Gamma_\mathcal{L}, \Delta_1, p : (P \Rightarrow\perp) \vdash \perp: *_p$	*Axiom of* $\lambda P-$ *and repeated weaken*
(e)	$\Gamma_\mathcal{L}, \Delta_1 \vdash (P \Rightarrow\perp) \Rightarrow\perp: *_p$	Π-form *on (c) and (d)*

For reasons of space and simplicity, we will omit these type derivations. Usually it will be evident that the types are correct.

9.2.2 Conversion for α-rules

Before we present the general scheme to convert α-rules, we describe the conversion of the typical case of an α-rule: conjunction. The tableau rule is:

$$\frac{P \wedge Q}{P, Q}$$

We have to derive $\Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash\ ? :\perp$. To the successor node, we assign the context $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P, q : Q$. By recursion, we get from this context a contradiction: $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P, q : Q \vdash c :\perp$. To derive a contradiction from the original context we use the structure of derivation 9.2.

$$
\begin{array}{ll}
 & \{ \text{ induction hypothesis } \} \\
(0) & \Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P, q : Q \vdash c :\perp \\
 & \{ \ \Pi\text{-}\textit{intro} \text{ on } (0) \ \} \\
(1) & \Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P \vdash (\lambda q : Q.c) : Q \Rightarrow\perp \\
 & \{ \ \Pi\text{-}\textit{intro} \text{ on } (1) \ \} \\
(2) & \Gamma_{\mathcal{L}}, \Delta_1, \Delta_2 \vdash (\lambda p : P.(\lambda q : Q.c)) : P \Rightarrow (Q \Rightarrow\perp) \\
 & \{ \text{ theorem 41 } \} \\
(3) & \Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash o : P \wedge Q \\
 & \{\wedge\text{-}\textit{elim}_1 \text{ on } (3) \ \} \\
(4) & \Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash \pi_1(o) : P \\
 & \{\wedge\text{-}\textit{elim}_2 \text{ on } (3) \ \} \\
(5) & \Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash \pi_2(o) : Q \\
 & \{ \text{ theorem 42 on } (2) \ \} \\
(6) & \Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash (\lambda p : P.(\lambda q : Q.c)) : P \Rightarrow (Q \Rightarrow\perp) \\
 & \{ \ \Pi\text{-}\textit{elim} \text{ on (4) and (6) } \} \\
(7) & \Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash (\lambda p : P.(\lambda q : Q.c))\ \pi_1(o) : Q \Rightarrow\perp \\
 & \{ \ \Pi\text{-}\textit{elim} \text{ on (5) and (7) } \} \\
(8) & \Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash (\lambda p : P.(\lambda q : Q.c))\ \pi_1(o)\ \pi_2(o) :\perp
\end{array}
$$

Derivation 9.2: Derivation structure for the α-rules.

Hence, the solution is given by $? := (\lambda p : P.(\lambda q : Q.c))\ \pi_1(o)\ \pi_2(o)$.

In the general case we consider the tableau rule:

$$\frac{E(P, Q)}{E_1(P), E_2(Q)}$$

We have to derive a contradiction from the context $\Gamma_{\mathcal{L}}, \Delta_1, o : E(P, Q), \Delta_2$. To the successor-node we assign the context $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : E_1(P), q : E_2(Q)$ from which we get a contradiction $c :\perp$ by recursion. In order to obtain a contradiction from the original context, we use a modified version of the scheme in derivation 9.2: Steps (0) till (2) remain unchanged, except that P has now become $E_1(P)$ and Q has become $E_2(Q)$. In step (3) we introduce $o : E(P, Q)$, but to continue with steps (4) till (8) we need $\Gamma_{\mathcal{L}}, \Delta_1, o : E(P, Q), \Delta_2 \vdash\ ?' : E_1(P) \wedge E_2(Q)$. How this is accomplished depends on the actual rule that is applied. For every rule we can construct a derivation and hence a λ-term to fill in for $?'$. The derivation of the individual λ-terms is omitted here, but the results are given in table 9.1. In this table, the conversion function T gives for a term $o : E(P,Q)$ a term with type $E_1(P) \wedge E_2(Q)$. Since steps (4) till (8) are performed after using the conversion function T, the appearances of o in these steps become $T(o)$. Note that the conversion functions produce λ-terms and that they are not λ-terms themselves.

$E(P,Q)$	$E_1(P)$	$E_2(Q)$	$T(o) : E_1(P) \wedge E_2(Q)$ with $o : E(P,Q)$
$P \wedge Q$	P	Q	o
$\neg(P \Rightarrow Q)$	P	$\neg Q$	$(\mathit{classic}\ P\ (\lambda p : \neg P.o(\lambda q : P.p\ q\ Q))$
			$, (\lambda q : Q.o(\lambda p : P.q)))$
$\neg(P \vee Q)$	$\neg P$	$\neg Q$	$(\lambda p : P.o\ (\mathit{injl}\ (P \vee Q)\ p)$
			$, \lambda q : Q.o\ (\mathit{injr}\ (P \vee Q)\ q))$

Table 9.1: Conversion functions for α-rules.

9.2.3 Conversion for β-rules

Again, we start with the typical case as an example. For β-rules the typical case is a disjunction, which has the tableau rule:

$$\frac{P \vee Q}{P \mid Q}$$

If the current context is $\Gamma_{\mathcal{L}}, \Delta_1, o : P \vee Q, \Delta_2$ then its successors will have contexts $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P$ and $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, q : Q$ respectively. From the successor contexts we have derived contradictions c_1 and c_2 by recursion. The derivation of a contradiction from the current context is then given by derivation 9.3.

To convert the general case we consider the tableau rule:

$$\frac{E(P,Q)}{E_1(P) \mid E_2(Q)}$$

We use the same strategy we used for α-rules: The derivation above is used as a scheme in which we have to replace P by $E_1(P)$ and Q by $E_2(Q)$ in lines (0) to (4). Instead of introducing $o : P \vee Q$ in line (5), we introduce $o : E(P,Q)$

{ induction hypothesis }
(0) $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P \vdash c_1 :\perp$
{ induction hypothesis }
(1) $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, q : Q \vdash c_2 :\perp$
{ Π-*intro* on (0) }
(2) $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2 \vdash (\lambda p : P.c_1) : P \Rightarrow\perp$
{ Π-*intro* on (1) }
(3) $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2 \vdash (\lambda q : Q.c_2) : Q \Rightarrow\perp$
{ $\vee$-*elim* on (2) and (3) }
(4) $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2 \vdash ((\lambda p : P.c_1)\nabla(\lambda q : Q.c_2)) : (P \vee Q) \Rightarrow\perp$
{ theorem 41 }
(5) $\Gamma_{\mathcal{L}}, \Delta_1, o : P \vee Q, \Delta_2 \vdash o : P \vee Q$
{ theorem 42 on (4) }
(6) $\Gamma_{\mathcal{L}}, \Delta_1, o : P \vee Q, \Delta_2 \vdash ((\lambda p : P.c_1)\nabla(\lambda q : Q.c_2)) : (P \vee Q) \Rightarrow\perp$
{ Π-*elim* on (5) and (6) }
(7) $\Gamma_{\mathcal{L}}, \Delta_1, o : P \vee Q, \Delta_2 \vdash ((\lambda p : P.c_1)\nabla(\lambda q : Q.c_2))\ o :\perp$

Derivation 9.3: Derivation structure for the β-rules.

and then insert a derivation between line (5) and line (6) that results in a λ-term of type $E_1(P) \vee E_2(Q)$. These λ-terms depend on o and can be obtained by applying a transformation function T to o. The transformation functions for β-rules are given in table 9.2 but their derivation is omitted. Again, the transformation functions T produce λ-terms but are not λ-terms themselves.

The remainder of the general case (the new lines (6) and (7)) then follows easily.

9.2.4 Conversion for γ-rules

In case of γ-rules the most typical example is the rule for universal quantification, with tableau rule:

$$\frac{\forall x : U.P}{\forall x : U.P,\ P^x_t}$$

Given the current context $\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2$ and the term t used to extend the tableau, we construct for the successor node the context

$E(P,Q)$	$E_1(P)$	$E_2(Q)$	$T(o) : E_1(P) \vee E_2(Q)$ with $o : E(P,Q)$
$\neg(P \wedge Q)$	$\neg P$	$\neg Q$	$classic\ (\neg P \vee \neg Q)\ \lambda r : \neg(\neg P \vee \neg Q).$ $r(injl\ (\neg P \vee \neg Q)\ (\lambda p : P.$ $r(injr\ (\neg P \vee \neg Q)\ (\lambda q : Q.o(p,q)))))$
$P \Rightarrow Q$	$\neg P$	Q	$classic\ (\neg P \vee Q)\ \lambda r : \neg(\neg P \vee Q).$ $r(injl\ (\neg P \vee Q)\ (\lambda p : P.$ $r(injr\ (\neg P \vee Q)\ (o\ p))))$
$P \vee Q$	P	Q	o

Table 9.2: Conversion functions for β-rules.

$\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2, p : P_t^x$. Note that the original universal quantifier is still present in this context. After the contradiction c has been derived from the successor's context by recursion we derive a contradiction from the original context according to derivation 9.4.

{ induction hypothesis }
(0) $\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2, p : P_t^x \vdash c : \perp$
{ Π-*intro* on (0) }
(1) $\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash (\lambda p : P_t^x.c) : P_t^x \Rightarrow \perp$
{ theorem 41 }
(2) $\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash o : (\forall x : U.P)$
{ ok because of theorem 38 on page 61 }
(3) $\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash t : U$
{ Π-*elim* on (2) and (3) }
(4) $\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash o\ t : P_t^x$
{ Π-*elim* on (1) and (4) }
(5) $\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash (\lambda p : P_t^x.c)\ (o\ t) : \perp$

Derivation 9.4: Derivation structure for the γ-rule.

To make this derivation suitable for the general case, consider the rule:

$$\frac{E(U,P)}{E(U,P), E'(P)_t^x}$$

We replace $(\forall x : U.P)$ by $E(U,P)$ and P_t^x by $E'(P)_t^x$ in the entire derivation. We then have to insert a derivation of a term of type $(\forall x : U.E'(P))$ from $o : E(U,P)$ after line (2). The resulting λ-term of this derivation is given by the transformation functions T given in table 9.3.

$E(U,P)$	$E'(P)$	$T(o) : (\forall x : U.E'(P))$ with $o : E(U,P)$
$(\forall x : U.P)$	P	o
$\neg(\exists x : U.P)$	$\neg P$	$(\lambda x : U.(\lambda p : P.o\ (inj\ (\exists x : U.P)\ p\ x)))$

Table 9.3: Conversion functions for γ-rules.

9.2.5 Conversion for δ-rules

The typical case for a δ-rule is existential quantification, with the tableau rule:

$$\frac{\exists x : U.P}{P^x_\theta},$$

where θ is a new variable of type U.

The current context is $\Gamma_{\mathcal{L}}, \Delta_1, o : (\exists x : U.P), \Delta_2$. For a δ-rule we have to extend the context more than for the other cases: we do not only add $p : P$ to the successor's context, but also a fresh variable $\theta : U$. The successor's context then reads $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, \theta : U, p : P$. By recursion we may assume that we have derived a contradiction c from this context. A contradiction from the current context is derived as shown in derivation 9.5.

{ induction hypothesis }
(0) $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, \theta : U, p : P \vdash c : \perp$
{ Π-*intro* on (0) }
(1) $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, \theta : U \vdash (\lambda p : P.c) : P \Rightarrow \perp$
{ Π-*intro* on (1) }
(2) $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2 \vdash (\lambda \theta : U.(\lambda p : P.c)) : (\forall \theta : U.(P \Rightarrow \perp))$
{ $\exists$-*elim* on (2) for $\perp$}
(3) $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2 \vdash \diamond\ (\exists x : U.P)\ (\lambda \theta : U.(\lambda p : P.c)) : (\exists x : U.P) \Rightarrow \perp$
{ theorem 41 }
(4) $\Gamma_{\mathcal{L}}, \Delta_1, o : (\exists x : U.P), \Delta_2 \vdash o : (\exists x : U.P)$
{ theorem 42 on (3) }
(5) $\Gamma_{\mathcal{L}}, \Delta_1, o : (\exists x : U.P), \Delta_2 \vdash \diamond\ (\exists x : U.P)\ (\lambda \theta : U.(\lambda p : P.c))$
$: (\exists x : U.P) \Rightarrow \perp$
{ Π-*elim* on (4) and (5) }
(6) $\Gamma_{\mathcal{L}}, \Delta_1, o : (\exists x : U.P), \Delta_2 \vdash (\diamond\ (\exists x : U.P)\ (\lambda \theta : U.(\lambda p : P.c)))\ o : \perp$

Derivation 9.5: Derivation structure for the δ-rule.

Like before, we use the above derivation to obtain a scheme for the general case. The tableau rule is:

$$\frac{E(U,P)}{E'(P)^x_\theta}$$

First, we replace P in the derivation above by $E'(P)^x_\theta$ in lines (0) to (2). In line (3), P is replaced by just $E'(P)$, which is allowed since the occurrences of x in P that were bound within $E(U,P)$ are now explicitly bound by the $\exists x : U \ldots$ occurring before $E'(P)$. Next, we change the *intro* in line (4) to an introduction of $o : E(U,P)$. Finally, we insert a derivation of a λ-term of type $(\exists x : U.E'(P))$ between line (4) and line (5). Also like before, these λ-terms are given by a transformation function T. The transformation functions for δ-rules are given in table 9.4.

$E(U,P)$	$E'(P)$	$T(o) : (\exists x : U.E'(P))$ with $o : E(U,P)$
$(\exists x : U.P)$	P	o
$\neg(\forall x : U.P)$	$\neg P$	$classic\ (\exists x : U.\neg P)\ (\lambda r : \neg(\exists x : U.\neg P).$ $o(\lambda x : U.classic\ P\ (\lambda p : \neg P.$ $r(inj\ (\exists x : U.\neg P)\ p\ x))))$

Table 9.4: Conversion functions for δ-rules.

9.3 Conversion of Closed Leaves

The recursive call ends when we convert a leaf of the tableau. However, at a closed leaf we have a context in which both a variable of type P and a variable of type $\neg P$ occur. In $\lambda P-$ negation is modeled by implication and $\bot$. Hence, in the context $\Gamma_{\mathcal{L}}, \Delta_1, p : P, \Delta_2, p' : \neg P, \Delta_3$ we can derive the contradiction $p'p$.

This concludes the conversion algorithm. Note that if during the construction of a tableau needless steps are taken these will also be translated.

9.4 Formalizing the Algorithm

The conversion algorithm can also be described as a function C from closed tableaux to λ-terms. Although this allows us to formally prove correctness of the conversion, we chose for the previous presentation since it is more descriptive in how the λ-terms are obtained. For the sake of completeness, we will now illustrate how the formal definitions are constructed.

Definition 59 (Conversion function C)
Let C :Closed Tableaux $\rightarrow$ Terms be the conversion function defined as

$$C(T) = classic\ P\ (\lambda p : \neg P.C'(\Gamma_{\mathcal{L}}, p : \neg P; T))$$

where $L(T) = \{\neg P\}$ is the label of the root of the tableau and C' :Contexts $\times$ Closed Tableaux $\rightarrow$ Terms is an auxiliary function to be defined next.

Definition 60 (Auxiliary function C')
The auxiliary function C' :(Contexts $\times$ Closed Tableaux) $\rightarrow$ Terms is defined recursively by distinction between the type of rule applied to the label of the root node of the tableaux. That is, $C'(con, tab)$ checks which rule was applied to the root node of tab. If tab consisted of a closed leaf, it returns a contradiction constructed as described in section 9.3. If a tableau rule was applied to tab it constructs contexts for resulting child nodes and recursively calls itself. The result from the recursive calls are combined to a single contradiction as described in section 9.2.

We can now state the correctness of the algorithm by the following theorems. We only give sketches of the proofs of these theorems, since the proofs can easily be extracted from the presentation of the algorithm.

Theorem 61 (Correctness of contradictions)
Let T be a closed tableau, let Γ be a valid context and define $L(T) = \{P \in \mathcal{P} \mid \exists p \in V.(p : P) \in \Gamma\}$ ($\mathcal{P}$ denotes the set of formulas of the logic $\mathcal{L}$). Then

$$\Gamma \vdash C'(\Gamma, T) : \perp .$$

Proof: By induction on the depth of the tableau. We will need cases for leaves and cases for nodes to which the special rule or one of the α-, β-, γ- or δ-rules is applied. It is easy to verify that the premises hold for the recursive function calls. $\square$

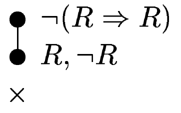
Theorem 62 (Correctness of conversion)
Let T be a closed tableau for P (i.e. $L(T) = \{\neg P\}$). Then

$$\Gamma_{\mathcal{L}} \vdash C(T) : P.$$

Proof: See 9.1 (converting the initial tableau) and use theorem 61. $\square$

9.5 Properties of Converted Tableaux

The converted proof may be much longer than a proof that is constructed directly in $\lambda P-$. For example: a direct proof of $R \Rightarrow R$ in $\lambda P-$ looks like $\Gamma_{\mathcal{L}} \vdash (\lambda p : R.p) : R \Rightarrow R$. However, if we convert the tableau



we get a much larger λ-term. Following the algorithm, we start with $\Gamma_{\mathcal{L}} \vdash$ *classic* $(R \Rightarrow R)$ $(\lambda o : \neg(R \Rightarrow R).c) : R \Rightarrow R$, where c is a contradiction extracted from the initial context $\Gamma_{\mathcal{L}}, o : \neg(R \Rightarrow R)$. The tableau rule applied is an α-rule for implication. The resulting λ-term of this conversion in general is $(\lambda p : E_1(P).(\lambda q : E_2(Q).c'))\ \pi_1(T(o))\ \pi_2(T(o)) :\perp$, in which c' is the contradiction derived from the successor's context $\Gamma_{\mathcal{L}}, p : E_1(P), q : E_2(Q)$. If we fill in P, Q, E_1, E_2 and T for our example and then use the result of this substitution in our proof, we get

$$\begin{aligned}&\mathit{classic}\ (R \Rightarrow R)\ (\lambda o : \neg(R \Rightarrow R).(\lambda p : R.\lambda q : \neg R.c')\\ &\pi_1(\mathit{classic}\ R\ (\lambda p : \neg R.o(\lambda q : R.pqR)), (\lambda q : R.o(\lambda p : R.q)))\\ &\pi_2(\mathit{classic}\ R\ (\lambda p : \neg R.o(\lambda q : R.pqR)), (\lambda q : R.o(\lambda p : R.q)))) : R \Rightarrow R\end{aligned}$$

where c' is the contradiction derived from the context $\Gamma_{\mathcal{L}}, p : R, q : \neg R$. This corresponds to the context in which the tableau gets closed by R and $\neg R$, hence the algorithm gives us $c' \equiv qp$. The final proof then reads:

$$\begin{aligned}\Gamma_{\mathcal{L}} \vdash\ &\mathit{classic}\ (R \Rightarrow R)\ (\lambda o : \neg(R \Rightarrow R).(\lambda p : R.\lambda q : \neg R.qp)\\ &\pi_1(\mathit{classic}\ R\ (\lambda p : \neg R.o(\lambda q : R.pqR)), (\lambda q : R.o(\lambda p : R.q)))\\ &\pi_2(\mathit{classic}\ R\ (\lambda p : \neg R.o(\lambda q : R.pqR)), (\lambda q : R.o(\lambda p : R.q)))) : R \Rightarrow R\end{aligned}$$

This 'explosion' of the proof term is certainly a drawback of this proof method. The size of the converted tableau can be decreased in several ways:

- One can use definitions or λ-abstraction/application pairs to avoid the repetition of sub-proofs in the converted tableau.
- It is possible to decrease the size of the tableau before it is converted. This is done by removing nodes from the tableau that only produce

formulas that are not needed to close the tableau. A straightforward algorithm to do this is easy to implement, but perhaps one should attempt to find a more advanced algorithm.

- If higher order logic is available, the translation constructs can be defined as polymorphic theorems in the context of the PTS. Instead of repeating the entire construct for every translated tableau-node, the translation then consists of a series of applications of these theorems.

- In an implementation one can choose not to convert the tableau at all. Instead, the following derivation rule is used:

$$\frac{\text{"a tableau for } A \text{ was found"}}{\Gamma \vdash \mathit{tab}\ A{:}A}$$

  However, this violates the safety of the system. It is therefore important that the tableau can be reconstructed and translated by the tool upon request. This approach is used in Cocktail.

Chapter 10

Combining Hoare Logic and PTSs

Our aim is to build an integrated tool for program derivation. However, a logic for program derivation as discussed in chapter 8 contains the rule of consequence, linking the programming language to validity statements about logical formulas. Therefore, it will be necessary to include a logical theorem prover into our programming tool.

Depending on the way the programming logic and the theorem prover logic are combined, there are certain restrictions and preliminaries on the tool. In this chapter, we discuss several methods to combine the programming logic with the proof logic and their consequences. Based upon this discussion, we will make a choice for our tool.

10.1 Deep embedding

The deep embedding of a programming language and its Hoare logic as described in this section was used by Homeier et al in [HM96]. In their paper, they discuss how a verification condition generator (VCG) is built within the theorem prover HOL (see [GM93]) using this embedding. A verification condition generator takes an annotated program as input and computes the theorems that have to be proved to guarantee that the program meets its specification. Even though our goal is to support program *derivation* rather than program *verification*, we will briefly discuss this kind of embedding. How to represent unfinished programs, needed during program derivation, using this embedding is not discussed.

The definition of *While* consists of the signature of the syntax and a function defining its denotational semantics based on state transitions. Hence, it is possible to define the language *While* in a PTS just like that: introduce a set $prog : *_s$ and define it to be all terms of a certain signature, by defining constructors for *prog*. In a PTS, this is done with the following context:

$$
\begin{array}{l}
prog : *_s, \\
skip : prog, \\
assign : var \to exp \to prog, \\
if : bool \to prog \to prog \to prog, \\
while : bool \to prog \to prog, \\
block : var \to exp \to prog \to prog, \\
comp : prog \to prog \to prog
\end{array}
$$

However, several constructors refer to other syntactic categories, like *var* and *exp* (for variables and expressions respectively). These can, in turn, be defined in the context as given in figure 10.1 before the definition of *prog*. For simplicity, we assume that the programming language only has variables of type *nat* and that *nat* is already defined in the logic with the usual operators.

For example, the expression $x + 3$ would be encoded as

$$plusE(varE(varx\ 0))(conE\ 3),$$

assuming that x is variable 0. The assignment $x := x + 3$ is denoted as

$$assign(varx\ 0)(plusE(varE(varx\ 0))(conE\ 3))$$

$var : *_s$,	
$varx : nat \to var$,	(creating an infinite amount of variables)
$exp : *_s$,	
$conE : nat \to exp$,	(creating numeric constants)
$varE : var \to exp$,	(creating expressions of single variables)
$plusE : exp \to exp \to exp$,	(representing addition)
$bool : *_s$,	
$equB : exp \to exp \to bool$,	(representing equality of expressions)
$smallerB : exp \to exp \to bool$,	(representing smaller than comparison)
$andB : bool \to bool \to bool$,	(representing boolean and)
$orB : bool \to bool \to bool$,	(representing boolean or)
$notB : bool \to bool$	(representing boolean not)

Figure 10.1: The context definitions used for deep embedding of the expressions of a simple programming language.

and the simple if statement **if** $x < 0$ **then** $x := x + 3$ **else skip** must be denoted as

$$\begin{aligned} &if\ (smallerB(varE(varx\ 0))(conE\ 0))\\ &\quad (assign(varx\ 0)(plusE(varE(varx\ 0))(conE\ 3)))\\ &\quad (skip)\end{aligned}$$

To define the denotational semantics, we need to formalize the concept of a state, which is a mapping from variables to values. Since, in this example, the language only uses natural numbers as values, a state is a function of type $var \to nat$. If we want to model other variable types as well, we need a set num of numerical values and use $var \to num$ as the type of a state.

If s is a state, we need to define $s[v \mapsto n]$ in order to denote the denotational semantics (see page 86). Since in a PTS it is not possible to define a higher order function that performs exactly this altering of a state, we must axiomatically define it. We will use the following relation

$Sub\ s1\ s2\ i\ n$,meaning that $s2$ is $s1[(varx\ i) \mapsto n]$ (we use natural number i to identify variables. The constructor $varx$ is then used to create the actual variable. Inherently, all variables are of the same type. If we want different types for variables, we have to change the type of i to *var* and define syntactic equivalence of variables, but this is beyond the scope of this thesis).

Hence, if we use *state* to denote the type $var \rightarrow nat$ then

$$Sub : state \rightarrow state \rightarrow nat \rightarrow nat \rightarrow *_p$$

which is defined as

$$\begin{aligned} Sub = \lambda s_1, s_2 : state.\lambda i, n : nat.\forall j : nat.(i = j) \Rightarrow s_2(varx\ j) = n \wedge \\ (i \neq j) \Rightarrow s_2(varx\ j) = s_1(varx\ j) \end{aligned}$$

Furthermore, we need the following relations which for simplicity are not given in PTS-notation:

$E\ e\ n\ s$	Stating that expression e evaluates to value n in state s.
$B\ b\ s$	Stating that boolean expression b evaluates to true in state s.
$C\ p\ s1\ s2$	Stating that program p, when started in state $s1$ yields state $s2$.

Since denotational semantics are not convenient to derive programs from specifications, it is necessary to define Hoare triples and prove some theorems about them.

$H\ pre\ p\ post$	Means that if program p is executed in a state $s1$ for which $pre\ s1$ holds, and p terminates in $s2$, then $post\ s2$ will hold, where pre and $post$ represent the precondition and postcondition respectively.

Hence, if we use *assert* to denote the type $state \rightarrow *_p$:

$$H : assert \rightarrow prog \rightarrow assert \rightarrow *_p,$$

and is defined as

$$\begin{aligned} H = \lambda pre : assert.\lambda p : prog.\lambda post : assert. \\ \forall s1 : state.(pre\ s1) \Rightarrow \forall s2 : state.C\ p\ s1\ s2 \Rightarrow (post\ s2) \end{aligned}$$

Probably, the simplest theorem to prove about H is the skip-theorem:

$$\forall a : assert.H\ a\ skip\ a$$

or the composition-theorem:

$$\begin{aligned} \forall a, c : assert.\forall p1, p2 : prog.(\exists b : assert.H\ a\ p1\ b \wedge H\ b\ p2\ c) \Rightarrow \\ (H\ a\ (comp\ p1\ p2)\ c) \end{aligned}$$

An embedding as presented above is called a deep embedding, since the programming language, the specification language and the denotational semantics are all embedded in the logic of the corresponding PTS. The Hoare logic consists of a set of theorems about the denotational semantics that are proved explicitly within the logic before they are used for program derivations. It is also possible to use an even deeper embedding in which the assertions used for the Hoare logic are defined by syntactic categories instead of being represented by predicates of the theorem prover's logic. A discussion of such an embedding is beyond the scope of this thesis.

Below, we discuss the advantages of deep embedding:

- The method follows exactly the definition of *While* as given in chapter 8, first defining the syntax, then the (denotational) semantics and finally the Hoare logic.
- Since the Hoare logic is formed by theorems proved within the theorem prover, it is sound by definition.
- One can prove completeness and other meta-theorems within the system. For instance, in [HM96], Homeier et al used it to prove correctness of a Hoare rule for mutually recursive procedures, which is not a trivial thing to do.

However, the method also has some drawbacks:

- Programs are encoded non-trivially in the logic, making them hard to read or write without specific tools. (For example, take a look at the examples given at the beginning of this section).
- Tools to construct programs that are encoded with a deep embedding will be difficult to write. Since the theorem prover logic is available to the user, she can add new programming constructs and derive new rules for the Hoare logic. Although this appears advantageous to the user, it means that the tool can encounter rules and programming construct that it was not designed to deal with.
- To use the Hoare logic, one has to formalize the denotational semantics of the programming language. Since it is already proved that the Hoare logic is sound and complete, this is unnecessary for a tool aimed at deriving programs. It proved to be useful, however, to design programming logics.

- The encoding requires higher order logic. For instance, an assertion is a predicate over a state, i.e. a predicate over a function. Automating the proving process will become very difficult.

To add functions and data types to the language, it is necessary to extend the syntax and define denotational semantics. While this is safe, the tool then needs to support those new types as well. This is not a trivial matter for a tool, since the new types may involve higher order logic.

To alleviate some of these drawbacks, one might choose a shallow embedding, which is discussed in the next section.

10.2 Shallow embedding

In a shallow embedding discussed in this section, due to von Wright et al in [vW94], the semantics of *While* is directly defined by its Hoare logic. The denotational semantics and the syntax of *While* are not formalized within the logic. Instead, programs are functions from predicates over states to predicates over states, based on Dijkstra's weakest precondition calculus. Weakest precondition calculus is a discipline on its own. We only discuss it briefly in this section. The interested reader is referred to [Dij76]. The weakest liberal precondition can considered to be a function wlp that computes for a given program S and postcondition Q the weakest predicate P for which $\{P\}S\{Q\}$ is a valid Hoare triple. I.e., $\{wlp(S,Q)\}S\{Q\}$ is a valid Hoare triple and $\{P\}S\{Q\}$ implies $P \Rightarrow wlp(S,Q)$.

For example:
$$\begin{array}{lcl} wlp(\mathbf{skip},Q) & = & Q, \\ wlp(x:=e,Q) & = & Q[x:=e], \\ wlp(S_1;S_2,Q) & = & wlp(S_1,wlp(S_2,Q)). \end{array}$$

In a shallow embedding, however, the weakest liberal precondition cannot be defined as above, since programs are no syntactical entities. Informally, a program S is now represented by S', where

$$S' = \lambda Q : state \rightarrow *_p . wlp(S,Q).$$

Hence, in the formalization, the program itself is a predicate transformer.

Formally, a shallow embedding of *While* is made as follows: like before, define var to be the set of variables and let $varx : nat \rightarrow var$ be its only constructor. Let $state = var \rightarrow nat$ be the type of states, which map variables to values and let $assert = state \to *_p$ be the type of predicates over states. Statements are terms of type $stat = assert \to assert$. Then *While*-statements are defined as follows:

$$\begin{array}{lcl} Skip & = & \lambda Q : assert.Q \\ Assign\ e & = & \lambda Q : assert.\lambda s : state.Q\ (e\ s) \\ If\, g\ p1\ p2 & = & \lambda Q : assert.\lambda s : state.(g\ s) \Rightarrow (p1\ Q\ s) \wedge \\ & & \qquad\qquad\qquad\qquad \neg(g\ s) \Rightarrow (p2\ Q\ s) \\ Comp\ p1\ p2 & = & \lambda Q : assert.p1\ (p2\ Q) \end{array}$$

where e is a function from *state* to *state* and g is an arbitrary predicate on states (i.e. g : *assert*). In order to define the while statement, a least fixed-point operator for assertions is needed, which in [vW94] is defined as follows:

$$fix = \lambda Q : stat.\lambda u : state.\forall p : assert.(\forall s : state.(Q\ p\ s) \Rightarrow (p\ s)) \Rightarrow (p\ u)$$

This fixed-point operator needs predicate transformers to be monotonic (see [vW94]). A predicate transformer is called monotonic if it returns a stronger precondition when applied to a stronger postcondition. All statements of *While* are monotonic.

Using the fixed-point operator, the while statement is defined as:

$$\begin{array}{lcl} while\ g\ p & = & \lambda Q : assert.fix\ \lambda a.assert.\lambda s : state.(g\ s) \Rightarrow (p\ a\ s) \wedge \\ & & \qquad\qquad\qquad\qquad\qquad\qquad \neg(g\ s) \to (Q\ s) \end{array}$$

Intuitively, fix computes the weakest assertion which implies the postcondition if the guard g is false and remains valid under execution of the body (program p) if the guard g is true.

To represent unfinished programs in a shallow embedding, one can define a magical statement that meets all specifications. In [vW94], this statement is called nondeterministic assignment. Its definition is

$$nondass\ m \ = \ \lambda Q.assert.\lambda s1 : state.\forall s2 : state.(m\ s1\ s2) \Rightarrow (Q\ s2)$$

where m is a relation between states. Operationally, the nondeterministic assignment, when executed in a state $s1$, will terminate in a state $s2$ for which $m\ s1\ s2$ holds. If several such states exist, one is chosen nondeterministically. If no such state exists, the statement will terminate in falsum. The *nondass* statement can be used as a placeholder for program parts that are not yet derived (for example, use $m = \lambda s1 : state.\lambda s2 : state.(P\ s1) \wedge (Q\ s2)$ to meet the specification with precondition P and postcondition Q).

Von Wright uses a shallow embedding in [vW94] to extend the theorem prover HOL with a mechanism for program derivation. This tool, called 'the refinement calculator', takes a specification of a program as input and solves it with a *nondass* statement. The user can then use tactics to focus on *nondass* statements in the program and refine them with other statements, possibly containing new *nondass* statements. This is not unlike theorem proving in theorem provers based on λ-calculi.

The advantages of a shallow embedding are as follows:

- The Hoare logic can directly be used to derive programs within the theorem prover.
- One can still prove some meta-theorems about programs, such as equality of certain program constructs. However, soundness and completeness of the logic can no longer be proved, since the denotational semantics are not formalized.
- Tools still have to create terms of the logic in order to represent programs; but now these terms are equal to terms representing the correctness proofs.
- In order to extend the language, one only has to define the corresponding predicate transformer.

Unfortunately, there are also some serious drawbacks:

- Since expressions are no longer syntactically defined, a user can use any assertion as guard in *if* and *while* statements. For instance, $(\forall p : nat.\exists q : nat.q > p \wedge prime(q))$ would be a valid guard, but is not allowed in any real-life programming language. While this might be useful to be allowed during program derivations, it is difficult to handle with a tool. The tool would then have to allow a different syntax for programs under construction than the syntax allowed in completed programs.
- Using arbitrary predicate transformers as programs may lead to solutions that are not implementable. For instance, a user may use a self-defined command that turns out to be inconsistent (e.g a *nondass* yielding false).

- If a function is used to specify a problem, there is no way to prohibit the use of this function to solve the problem (for instance, the program $\{N \geq 0\}x := \mathit{fib}(N)\{x = \mathit{fib}(N)\}$ is correct, but undesirable). In deep embedding, this is not possible, since the programming language's syntax is explicitly defined and does not contain the function *fib*.

- Shallow embedding still requires higher order logic and hence, automatic proof search will be limited.

For building a user-friendly tool, both embeddings have common drawbacks: (1) The tool has to provide the programmer with a user interface for a theory (Hoare logic) formalized within a theorem prover. Since the theorem prover allows extensions of the formalization, the tool has to be able to cope with all these extensions as well. The tool can therefore not assume any limits about the creativity of the user, let alone impose restrictions on it (the restrictions are set by the theorem prover used for formalization). Since the theorem prover is needed to construct proofs during program derivation, it is also not possible to shield its use from the programmer. (2) Both embeddings require higher order logic. In this logic, simple notions like $\wedge$ and $\vee$ are usually encoded by higher order constructs and therefore, automatic proof search will be very limited. As a result, the programmer has to prove many trivial theorems by hand.

Therefore, we consider yet another way to combine Hoare logic with a theorem prover in the next section. However, this time we will not formalize the Hoare logic within a theorem prover.

10.3 No embedding

In the previous two sections, Hoare logic was formalized within a theorem prover. This required higher order logic and hence, automatic proof search became very difficult. However, in an environment for program derivation, one will often encounter proof obligations that are very simple. Hence, in practice, the user will have a need for automatic support to solve these proof obligations.

Also, we want the tool to focus on the program rather than its correctness proof. It should also be possible for the user to enter a program without its specification or to postpone the construction of a correctness proof till the program is completed. Hence, we do not want the programming process be dependent on the proving process.

In this section, we will link a Hoare logic to a theorem prover without embedding it. Instead, we will create derivation rules for programs, that depend on derivations in a PTS. This allows us to restrict the PTS to first-order logic and support automated theorem proving as described in the chapters 7 and 9.

The only rule in the Hoare logic of *While* that refers to theorems, is the rule of consequence:

$$\frac{\models P' \Rightarrow P \quad \{P\}S\{Q\} \quad \models Q \Rightarrow Q'}{\{P'\}S\{Q'\}}$$

Stated this way, one assumes $\models$ to denote semantical validity of formulas in a logic, which is left implicit. This logic, however, must be powerful enough to deal with all possible expressions allowed in the language. This is usually referred to as the expressibility requirement.

Since, in our definition of *While*, the expressions in the programs are those defined in the logic, we automatically fulfill the expressibility requirement. In case of first-order logic, one can use $\lambda P-$ to construct the required proofs. The rule of consequence then becomes:

$$\frac{\Gamma_{\mathcal{L}} \vdash p{:}P' \Rightarrow P \quad \{P\}S\{Q\} \quad \Gamma_{\mathcal{L}} \vdash q{:}Q \Rightarrow Q'}{\{P'\}S\{Q'\}},$$

where L is the logic used to define *While* and $\Gamma_{\mathcal{L}}$ is the corresponding context for $\lambda P-$ as defined in definition 36 on page 61.

The advantages of this approach are:

- Programs are directly accessible by tools, since they are syntactical terms themselves.
- Boolean expressions allowed in programs are defined within the logic, but are separated from the specification language (e.g. one cannot use $(\forall p : nat.\exists q : nat.q > n \wedge prime(q))$ as a guard, since it is a propositional formula, not a boolean).
- The logic can be restricted to first-order logic and hence, meaningful automatic proof search is possible.

Even though this "new" rule of consequence is sufficient to implement a sound and complete Hoare logic, it still suffers from several drawbacks we encountered in the embeddings:

- Programs cannot be checked after they have been constructed. There is no term representing a correctness proof of the entire program.
- There is no way to prevent the usage of specification functions in programs. For instance, $x := \mathit{fib}(N)$ is a valid, although undesirable, program.

In the following, we show how these drawbacks can be alleviated.

Programs cannot be checked for correctness after they have been constructed, because the proofs used for application of the rule of consequence are usually not stored within the program and cannot be (re)constructed automatically. Since proofs are syntactically represented in λ-calculus, we can easily incorporate those proofs by extending the program-syntax and change the rule of consequence to something like:

$$\frac{\Gamma_{\mathcal{L}} \vdash p{:}P' \Rightarrow P \quad \{P\}S\{Q\} \quad \Gamma_{\mathcal{L}} \vdash q{:}Q \Rightarrow Q'}{\{P'\}\mathbf{cons}\ p\ S\ q\{Q'\}}$$

However, during program derivation, one usually alters only the precondition or the postcondition, not both at once. Also, since program S now has become embedded in *cons* p S q, it is less accessible to the tool. If one regards the change of P to P' as a re-formulation of a state-property, one could consider application of the rule of consequence to be an application of the theorem $P' \Rightarrow P$ to a state in which P holds. The same can be said about $Q \Rightarrow Q'$. We denote this application of a theorem by the program **fake** p, which has the same denotational semantics as **skip**, since the state does not change. The rule of consequence can now be replaced by the simpler rule:

$$\frac{\Gamma_{\mathcal{L}} \vdash p{:}P \Rightarrow Q}{\{P\}\mathbf{fake}\ p\{Q\}}$$

The original rule of consequence can now be derived as follows: From $\Gamma_{\mathcal{L}} \vdash p{:}P' \Rightarrow P$ and $\Gamma_{\mathcal{L}} \vdash q{:}Q \Rightarrow Q'$ we respectively derive $\{P'\}\mathbf{fake}\ p\{P\}$ and $\{Q\}\mathbf{fake}\ q\{Q'\}$. Since $\{P\}S\{Q\}$, we use the composition rule to conclude

$$\{P'\}\mathbf{fake}\ p; S; \mathbf{fake}\ q\{Q'\}.$$

The fake-statement has the advantage that it allows separate treatment of pre- and postconditions. Also, all proofs are now stored in separate statements, not having other programs as sub-programs.

Using *While*, extended with fake-statements, yields programs that can be checked once they are constructed. This is nearly a trivial matter, since every statement can only be derived by a single rule, including the fake-statement. The premise of the fake-statement could also read $\Gamma \vdash p{:}P \Rightarrow Q$ for any other PTS, as long as this type judgment can be checked automatically.

However, this "type-checking" for programs has limited applicability: One could for instance derive a program

$$\{a = 4\}\textbf{fake } p; a := a + 1\{a = 5\},$$

where p is a proof of $(a = 4) \Rightarrow (a + 1 = 5)$. However, checking

$$\{a \geq 0\}\textbf{fake } p; a := a + 1\{a \geq 1\}$$

would fail, since the proof p stored in the fake-statement has the wrong type; instead we need a proof q of $(a \geq 0) \Rightarrow (a + 1 \geq 1)$.

Since our tool only needs to check if programs meet the specification for which they were derived, this is an acceptable restriction.

Having proofs explicitly stated in programs seems unnatural. However, this is not necessarily true, since one can consider programs to be proofs of the satisfiability of their specification. From this point of view, programs are the λ-terms of a Hoare logic. Since, in our tool, the Hoare logic is linked to a PTS this view is also more consistent with the formalism used for proofs. Therefore, we will introduce a different notation for programs and their specifications.

Definition 63 (Program Specification)
Let P and Q be a pre- and postcondition respectively. Then $P \rhd Q$ is a program specification. The Hoare triple $\{P\}S\{Q\}$ can now be denoted as $S : P \rhd Q$, stating that program S satisfies specification $P \rhd Q$.

The reason that programs like $x := \mathit{fib}(N)$ are allowed, is that programs are based on exactly the same logic as specifications. Therefore, all function-symbols and expressions available in the specification are also available in programs.

However, in a PTS like $\lambda P-$, all function symbols of a logic L are explicitly declared within the context $\Gamma_{\mathcal{L}}$. If we add contexts to the Hoare logic, we can use a larger context for specifications than for programs. The context accessible from the program should always be part of the context accessible from the specification though, since we might get expressibility problems otherwise. For instance, consider a function *sqr*, computing the square of a natural number, that only exists in the programming language context and not in the specification language context. Then the precondition of the assignment $x := sqr(2)$ with respect to the postcondition $x = 4$ reads $sqr(2) = 4$, but cannot be expressed in the specification.

Therefore, we split contexts for Hoare triples into three parts:

- The first part is accessible from programs as well as specifications. Programs can use this context, but not alter its variables. Typically, it contains all function symbols, constants and definitions that are default to the language (e.g. the type **bool** of booleans). This context is referred to as the *language context*.

- The second part contains (locally) defined program variables that can be altered by programs[1]. This context can depend on the first context, e.g. a program variable could have pre-defined type **bool**. Typically, this context is used to store constants and variables needed to specify a programming problem, for instance the variable x from the postconditions $x = fib(N)$. This context is called the *program context*.

- The third context contains all other logical elements needed to specify programs, like abstract data types or auxiliary functions. This context cannot be used by programs, only by specifications. It may depend on both previous contexts, since a postcondition may specify that some language expression must be equal to an auxiliary function (e.g. $x = fib(N)$, where *fib* is an element of the third context and hence, cannot be used by the program). This context is referred to as the *specification context*.

The order of these contexts is quite natural: there would be little need for functions and variables accessible only from programs and not from specifications.

[1]The second context can also contain constants, function symbols etc, but by the definition of the abstract syntax of programs, these can never be altered by the program.

Hence, we will add triples of contexts to Hoare triples in the following manner:

Definition 64 (Hoare Contexts)
Let Γ_1, Γ_2 and Γ_3 be contexts of a CPTS as described in definition 16 on page 48, then $\langle\Gamma_1, \Gamma_2, \Gamma_3\rangle$ is a Hoare context.

Note the following important differences:

1. $\langle\Gamma_1, \Gamma_2, \Gamma_3\rangle$ is a triple of contexts, hence a Hoare context.
2. Since Γ_1, Γ_2 and Γ_3 are contexts of CPTSs, Γ_1, Γ_2 (the concatenation of Γ_1 and Γ_2, denoted without the triple-brackets) and $\Gamma_1, \Gamma_2, \Gamma_3$ (concatenation again) are CPTS-contexts too.

Definition 65 (Hoare logic with explicit contexts)
Let $\vdash_\lambda$ be the type-judgment relation of a CPTS. Then we define $\vdash_H$ to be a Hoare derivation system defined by the rules shown in figure 10.2.

We briefly comment on the use of the context for each rule:

Spec This rule repeats the definition of a specification. It could be eliminated, but is used for consistency with the formal definition of PTSs. Note that the pre- and postcondition may depend on the entire context.

Skip In order for **skip** : $P \triangleright P$ to hold, $P \triangleright P$ must be a well-formed specification.

Assign For $x := e : P[x := e] \triangleright P$ to hold, $P[x := e] \triangleright P$ must be a specification and e must be an expression of the same set-type as variable x. Moreover, x must occur in the program context and e may only depend on the language and program context.

If The first premise claims that e is a boolean expression that can be derived from the language and program context. The other premises are direct translations from the original Hoare logic.

While Similar to *If*.

[*Spec*]
$$\frac{\Gamma_1,\Gamma_2,\Gamma_3 \vdash P{:}*_p \quad \Gamma_1,\Gamma_2,\Gamma_3 \vdash Q{:}*_p}{\langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash P \triangleright Q{:}\mathbf{Spec}}$$

[*Skip*]
$$\frac{\langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash P \triangleright P{:}\mathbf{Spec}}{\langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash \mathbf{skip}{:}P \triangleright P}$$

[*Assign*]
$$\frac{\begin{array}{l}\Gamma_1,(\Delta_1,x:U,\Delta_2) \vdash U{:}*_s \\ \Gamma_1,(\Delta_1,x:U,\Delta_2) \vdash e{:}U \\ \langle\Gamma_1,(\Delta_1,x:U,\Delta_2),\Gamma_3\rangle \vdash P[x:=e] \triangleright P{:}\mathbf{Spec}\end{array}}{\langle\Gamma_1,(\Delta_1,x:U,\Delta_2),\Gamma_3\rangle \vdash x:=e{:}P[x:=e] \triangleright P}$$

[*If*]
$$\frac{\begin{array}{l}\Gamma_1,\Gamma_2 \vdash e{:}\mathbf{bool} \\ \langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash S_1{:}P \wedge e = \mathbf{true} \triangleright Q \\ \langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash S_2{:}P \wedge e = \mathbf{false} \triangleright Q\end{array}}{\langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}{:}P \triangleright Q}$$

[*While*]
$$\frac{\begin{array}{l}\Gamma_1,\Gamma_2 \vdash e{:}\mathbf{bool} \\ \langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash S{:}P \wedge e = \mathbf{true} \triangleright P\end{array}}{\langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash \mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{od}{:}P \triangleright P \wedge e = \mathbf{false}}$$

[*Block*]
$$\frac{\begin{array}{ll}\Gamma_1,\Gamma_2 \vdash U{:}*_s & \\ \Gamma_1,\Gamma_2 \vdash e{:}U & x \text{ is } \Gamma_1,\Gamma_2,\Gamma_3\text{-fresh} \\ \langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash P \triangleright Q{:}\mathbf{Spec} & \\ \langle\Gamma_1,(\Gamma_2,x:U),\Gamma_3\rangle \vdash S{:}P \wedge x = e \triangleright Q & \end{array}}{\langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash |[\mathbf{var}\ x:=e:U\ \bullet\ S\ ]|{:}P \triangleright Q}$$

[*Comp*]
$$\frac{\begin{array}{l}\langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash S_1{:}P \triangleright Q \\ \langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash S_2{:}Q \triangleright R\end{array}}{\langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash S_1;S_2{:}P \triangleright R}$$

[*Fake*]
$$\frac{\Gamma_1,\Gamma_2,\Gamma_3 \vdash p{:}P \Rightarrow Q}{\langle\Gamma_1,\Gamma_2,\Gamma_3\rangle \vdash \mathbf{fake}\ p{:}P \triangleright Q}$$

Figure 10.2: A Hoare logic with explicit contexts.

Block Contexts play a main role here. The first two premises claim that e is an expression of some set-type available to the program (i.e. both U and e are derived from only language and program context). $P \rhd Q$ must be a valid specification in the full context; but without the fresh variable x. S is a program, which may use a fresh variable x in its program context and which satisfies $P \wedge x = e \rhd Q$.

Comp This rule is a direct translation of the original rule from the Hoare logic.

Fake The *Fake* rule was explained before. Note that since $P \Rightarrow Q$ is a proposition, so are P and Q. Hence, the program is correctly specified.

The Hoare logic now has a notation and a set of derivation rules similar to those of a pure type system. Through the following theorems, we will prove that the Hoare logic also has some important meta-theoretical properties in common with the PTSs: programs can be checked for correctness once they are completed. This also enables the communication of those programs: programs which include **fake**-statement are self-contained and require no further proof of correctness.

Definition 66
Given a Hoare logic with explicit contexts as defined above and a context $\Gamma \equiv \langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle$*, we define the following concepts:*

Program synthesis: Given precondition P and postcondition Q, automatically find a program S, such that $\Gamma \vdash S{:}P \rhd Q$.

Backward inference: Given a program S and a postcondition Q, automatically find a precondition P, such that $\Gamma \vdash S{:}P \rhd Q$.

Forward inference: Given a program S and a precondition P, automatically find a postcondition Q, such that $\Gamma \vdash S{:}P \rhd Q$.

Specification inference: Given a program S, automatically find a precondition P and a postcondition Q, such that $\Gamma \vdash S{:}P \rhd Q$.

Program checking: Given precondition P, postcondition Q and program S, automatically verify if $\Gamma \vdash S{:}P \rhd Q$.

Program synthesis is decidable, but not useful. This may sound contradictory, since program synthesis appears very desirable, but the property is decidable through the existence of a trivial solution.

Theorem 67
Program synthesis is decidable.

Proof: For an arbitrary precondition P and postcondition Q, the program

$$S \equiv \textbf{fake}\ p; \textbf{while true do skip od}; \textbf{fake}\ q,$$

has specification $P \triangleright Q$, where p and q represent proofs of $P \Rightarrow (\neg \perp)$ and $(\neg \perp \wedge \textbf{true} = \textbf{false}) \Rightarrow Q$ respectively, which can easily be constructed. $\square$

Clearly, this program may be theoretically correct, but it does not solve the programming problem at hand, since it does not terminate.

Useful solutions cannot be found fully automatically. For example, consider a precondition P which implies a postcondition Q. A useful solution would be the statement **fake** p, where p is a proof of $P \Rightarrow Q$, but it is known that, in general, such a proof cannot be generated fully automatically.

Theorem 68 (Backward inference is decidable)
Let Γ be a Hoare context, let S be a program and let Q be a postcondition. One can compute a unique P such that $\Gamma \vdash S{:}P \triangleright Q$ if it exists.

Proof: Let $\Gamma \equiv \langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle$ denote a Hoare context. We use induction on the structure of S:

case $S \equiv$ **skip:** Use $P \equiv Q$.

case $S \equiv x := e$**:** Use $P \equiv Q[x := e]$.

case $S \equiv$ **if** g **then** S_1 **else** S_2 **fi:** By induction, compute P_1 and P_2, such that $\Gamma \vdash S_1{:}P_1 \triangleright Q$ and $\Gamma \vdash S_2{:}P_2 \triangleright Q$. If P_1 has the form $R \wedge g = \textbf{true}$ and P_2 has the form $R \wedge g = \textbf{false}$ then $P \equiv R$ will suffice. Otherwise, no solution exists since the Hoare logic is purely syntax-driven. To change an assertion into an equivalent one that is syntactically different, a **fake**-statement must be used.

case $S \equiv \textbf{while } g \textbf{ do } S_1 \textbf{ od}$**:** In order for P to exist, Q must have the form $I \wedge g = \textbf{false}$. By induction we can compute P_1, such that $\Gamma \vdash S_1{:}P_1 \rhd I$. P_1 must have the form $I \wedge g = \textbf{true}$. If so, choose $P \equiv I$. Otherwise, no solution exists.

case $S \equiv |[\textbf{var } x := e : U \bullet S_1]|$**:** x may not occur free in Q. By induction hypothesis, compute P_1 such that $\langle \Gamma_1, (\Gamma_2, x : U), \Gamma_3 \rangle \vdash S_1{:}P_1 \rhd Q$. Now P_1 must have the form $R \wedge x = e$, where x does not occur free in R. If so, choose $P \equiv R$. Otherwise, no solution exists.

case $S \equiv S_1; S_2$**:** By induction hypothesis, compute P_2 with $\Gamma \vdash S_2{:}P_2 \rhd Q$. Then compute P_1 such that $\Gamma \vdash S_1{:}P_1 \rhd P_2$. If both computations succeed, choose $P \equiv P_1$. Otherwise no solution exists.

case $S \equiv \textbf{fake } p$**:** From type theory it is known how to construct T_p, such that $\Gamma_1, \Gamma_2, \Gamma_3 \vdash p{:}T_p$. T_p must have the form $R \Rightarrow Q$. If so, choose $P \equiv R$. Otherwise, no solution exists.

□

Corollary 69 (Program checking is decidable)
From theorem 68 it immediately follows that given a Hoare context Γ*, a program* S *and a specification* $P \rhd Q$*, it is decidable whether or not* $\Gamma \vdash S{:}P \rhd Q$*.*

In the following theorems we will sometimes need the set of postconditions that correspond to a certain precondition. Since programs may contain assignments, we need to invert the substitutions performed by these assignments. Therefore we define inverse substitutions.

Definition 70 (Inverse substitution)
Let ϕ *be a mapping from variables to expressions (i.e. a substitution). We define* ϕ^{-1} *as the function from propositions to sets of propositions by:*

$$\phi^{-1}(P) = \{Q \mid P \equiv \phi(Q)\}$$

Without further proof we claim that ϕ^{-1} *is computable and that* $\phi^{-1}(Q)$ *is finite for every* Q*, provided that* $\phi(x) \neq x$ *only for a finite number of known variables* x*.*

For example, consider the substitution $\phi \equiv \{x \mapsto 30\}$. Then $\phi^{-1}(r = 30 + 30) = \{r = 30 + 30, r = x + 30, r = 30 + x, r = x + x\}$ and $\phi^{-1}(r =$

$x + 1) = \emptyset$. For $\psi \equiv \{x \mapsto x + 1\}$, we get $\psi^{-1}(x + 1 < 8) = \{x < 8\}$ and $\psi^{-1}(x + 3 < 6) = \emptyset$. Note that it is important to know the domain of the substitution and that this domain is finite if we want to compute the inverse substitution.

Theorem 71 (Forward inference is decidable)
Given a Hoare context Γ, a program S and a precondition P, one can compute a set Q^s of propositions, such that

$$Q \in Q^s \textit{ if and only if } \Gamma \vdash S{:}P \rhd Q.$$

From this it immediately follows that forward inference is computable.

Proof: Let $\Gamma \equiv \langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle$ denote a Hoare context. We use induction on the structure of S:

case $S \equiv$ **skip:** Use $Q^s \equiv \{P\}$.

case $S \equiv x := e$**:** Compute $Q^s \equiv \phi^{-1}(P)$, where $\phi = \{x \mapsto e\}$.

case $S \equiv$ **if** g **then** S_1 **else** S_2 **fi:** By induction, compute sets Q_1^s and Q_2^s, such that for each $Q_1 \in Q_1^s$ we have $\Gamma \vdash S_1{:}P \wedge g = \textbf{true} \rhd Q_1$ and for each $Q_2 \in Q_2^s$ we have $\Gamma \vdash S_2{:}P \wedge g = \textbf{false} \rhd Q_2$. Since S can only be derived if $Q_1 \equiv Q_2$, choose $Q^s \equiv Q_1^s \cap Q_2^s$.

case $S \equiv$ **while** g **do** S_1 **od:** By induction, compute Q_1^s such that for each $Q_1 \in Q_1^s$ we have $\Gamma \vdash S_1{:}P \wedge g = \textbf{true} \rhd Q_1$. Since S can only be derived if $P \in Q_1^s$, choose $Q^s \equiv \{P \wedge g = \textbf{false}\}$ if $P \in Q_1^s$ and $Q^s \equiv \emptyset$ otherwise.

case $S \equiv |[\textbf{var}\ x := e : U \bullet S_1]|$**:** If x occurs free in P choose $Q^s \equiv \emptyset$. Otherwise, compute by induction hypothesis Q_1^s such that for each $Q_1 \in Q_1^s$ we have $\langle \Gamma_1, (\Gamma_2, x : U), \Gamma_3 \rangle \vdash S_1{:}P \wedge x = e \rhd Q_1$. Since x may not occur in the postcondition of S, choose $Q^s \equiv \{Q \in Q_1^s \mid x \notin FV(Q)\}$.

case $S \equiv S_1; S_2$**:** By induction hypothesis, compute Q_1^s such that for each $Q_1 \in Q_1^s$ with $\Gamma \vdash S_1{:}P \rhd Q_1$. For each $Q_1 \in Q_1^s$, compute the set $Q_{Q_1}^s$ such that for each $Q_2 \in Q_{Q_1}^s$ we have $\Gamma \vdash S_2{:}Q_1 \rhd Q_2$. Choose $Q^s \equiv \bigcup_{Q_1 \in Q_1^s} Q_{Q_1}^s$.

case $S \equiv$ **fake** p**:** From type theory it is known how to construct T_p, such that $\Gamma_1, \Gamma_2, \Gamma_3 \vdash p{:}T_p$. T_p must have the form $P \Rightarrow Q$. If so, choose $Q^s \equiv \{Q\}$. Otherwise, choose $Q^s \equiv \emptyset$.

□

Lemma 72 (Specification inference with fake)
Let Γ be a Hoare context and S be a program containing at least one **fake**-*statement. It is possible to compute a unique proposition P and a set of propositions Q^s, such that*

$$Q \in Q^s \textit{ if and only if } \Gamma \vdash S{:}P \triangleright Q.$$

Proof: Let $\Gamma \equiv \langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle$ denote a Hoare context. We use induction on the structure of S:

case $S \equiv$ **skip:** Since **skip** contains no **fake** statement, this case does not occur.

case $S \equiv x := e$**:** See **skip**.

case $S \equiv$ **if** g **then** S_1 **else** S_2 **fi:** Without loss of generality, assume that S_1 contains a **fake**-statement. Compute P_1 and Q_1^s, such that for every $Q_1 \in Q_1^s$ we have $\Gamma \vdash S{:}P_1 \triangleright Q_1$. If P_1 is not of the form $R \wedge g =$ **true**, choose $P \equiv P_1$ and $Q^s \equiv \emptyset$. Otherwise, use Forward inference (theorem 71) to compute Q_2^s, such that for every $Q_2 \in Q_2^s$ we have $\Gamma \vdash S_2{:}R \wedge g = \textbf{false} \triangleright Q_2$. Since S can only be derived if $Q_1 \equiv Q_2$, choose $P \equiv R$ and $Q^s \equiv Q_1^s \cap Q_2^s$.

case $S \equiv$ **while** g **do** S_1 **od:** By induction, compute P_1 and Q_1^s such that for each $Q_1 \in Q_1^s$ we have $\Gamma \vdash S_1{:}P_1 \triangleright Q_1$. If P_1 is not of the form $I \wedge g =$ **true**, choose $P \equiv P_1$ and $Q^s \equiv \emptyset$. Otherwise, choose $P \equiv I$. Since S can only be derived if $I \in Q_1^s$, choose $Q^s \equiv \{I \wedge g = \textbf{false}\}$ if $I \in Q_1^s$ and $Q^s \equiv \emptyset$ otherwise.

case $S \equiv |[\textbf{var}\ x := e : U \ \bullet\ S_1]|$**:** From the induction hypothesis we compute P_1 and Q_1^s such that we have $\langle \Gamma_1, (\Gamma_2, x : U), \Gamma_3 \rangle \vdash S_1{:}P_1 \triangleright Q_1$ for each $Q_1 \in Q_1^s$. If P_1 is not of the form $R \wedge x = e$ where x does not occur free in R, choose $P \equiv P_1$ and $Q^s \equiv \emptyset$. Otherwise, choose $P \equiv R$. Since x may not occur free in the postcondition of S, choose $Q^s \equiv \{Q \in Q_1^s \mid x \notin FV(Q)\}$.

case $S \equiv S_1; S_2$**:** If S_1 contains a **fake**-statement, use the induction hypothesis to compute P_1 and Q_1^s, such that for each $Q_1 \in Q_1^s$ we have $\Gamma \vdash S_1{:}P_1 \triangleright Q_1$. For each $Q_1 \in Q_1^s$, use Forward inference (theorem 71) to compute the set $Q_{Q_1}^s$ such that for each $Q_2 \in Q_{Q_1}^s$ we have $\Gamma \vdash S_2{:}Q_1 \triangleright Q_2$. Choose $P \equiv P_1$ and $Q^s \equiv \bigcup_{Q_1 \in Q_1^s} Q_{Q_1}^s$.

If S_2 contains a **fake**-statement, use the induction hypothesis to compute P_2 and Q_2^s, such that for each $Q_2 \in Q_2^s$ we have $\Gamma \vdash S_2{:}P_2 \triangleright Q_2$. Use Backward inference (theorem 68) to compute the precondition P_1

such that $\Gamma \vdash S_1{:}P_1 \triangleright P_2$ if it exists. If it does not exist, choose $P \equiv P_2$ and $Q^s \equiv \emptyset$. Otherwise, choose $P \equiv P_1$ and $Q^s \equiv Q^s_2$.

case $S \equiv$ **fake** p**:** From type theory it is known how to construct T_p, such that $\Gamma_1, \Gamma_2, \Gamma_3 \vdash p{:}T_p$. T_p must have the form $R \Rightarrow Q$. If so, choose $P \equiv R$ and $Q^s \equiv \{Q\}$. Otherwise, choose $P \equiv T_p$ and $Q^s \equiv \emptyset$.

□

What remains is to compute specifications for *While* programs without **fake** statements. In all previous inference theorems, the structure of the program specification was derived from the types of the proofs stored in **fake**-statements. Since this is no longer possible, we define the relation between pre- and postcondition in such a way that we can search for specifications with this relation.

From the rule *Assign*, we can already see that the precondition depends on the postcondition using a substitution. Also, we can see from the rules for *Block*, *If* and *While* that the postcondition will have a more complex structure than the precondition, since we have no **fake**-statements to simplify the formulas (the Hoare logic for *While* is purely syntax based!). Also, the specification must meet certain constraints (for instance, to derive an **if**-statement, both premisses must have the same precondition).

These ingredients together form the basis of the complex relation between pre- and postcondition of a program, given in theorem 75. Therefore, we will first define propositional patterns and a notion of constraints.

A propositional pattern is a series of left-associative conjuncts with a placeholder at the bottom-left of its tree-representation. Hence, if the placeholder is replaced by a propositional pattern, we get a new propositional pattern. If it is replaced by a proposition, we get a proposition.

Definition 73 (Propositional patterns)
The set of propositional patterns is defined as:

- *The special symbol α is a propositional pattern.*
- *If $\mathcal{P}$ is a propositional pattern and Q is a proposition, then $(\mathcal{P} \wedge Q)$ is a propositional pattern.*

We also define the application of a pattern $\mathcal{P}$ to a propositional pattern or a proposition X as:

$$\begin{array}{ll} \mathcal{P}(X) \equiv X & \textit{if } \mathcal{P} \equiv \alpha \\ \mathcal{P}(X) \equiv \mathcal{P}_1(X) \wedge P_2 & \textit{if } \mathcal{P} \textit{ has the form } \mathcal{P}_1 \wedge P_2 \end{array}$$

Note that if X is a pattern, $\mathcal{P}(X)$ is also a pattern and if X is a proposition, $\mathcal{P}(X)$ is a proposition. For sets $\mathcal{P}^s$ of patterns we define

$$\begin{array}{lcl} \mathcal{P}^s(X) & = & \{\mathcal{P}(X) \mid \mathcal{P} \in \mathcal{P}^s\} \textit{ and} \\ \mathcal{P}^s(X^s) & = & \{\mathcal{P}(X) \mid \mathcal{P} \in \mathcal{P}^s, X \in X^s\} \end{array}$$

Also, define for pattern $\mathcal{P}$ and substitution ϕ that

$$\begin{array}{ll} \phi(\mathcal{P}) \equiv \alpha & \textit{if } \mathcal{P} \equiv \alpha \\ \phi(\mathcal{P}) \equiv \phi(\mathcal{P}_1) \wedge \phi(P_2) & \textit{if } \mathcal{P} \textit{ has the form } \mathcal{P}_1 \wedge P_2 \end{array}$$

For sets $\mathcal{P}^s$ of patterns we define $\phi(\mathcal{P}^s) = \{\phi(\mathcal{P}) \mid \mathcal{P} \in \mathcal{P}^s\}$.

For example, let $((\alpha \wedge P_1) \wedge P_2)$ be a pattern. Then $((\alpha \wedge P_1) \wedge P_2)(P_0)$ yields the proposition $((P_0 \wedge P_1) \wedge P_2)$. When applied to $(\alpha \wedge P_0)$ we get propositional pattern $(((\alpha \wedge P_0) \wedge P_1) \wedge P_2)$

Definition 74 (Constraints)
A constraint C is a pair of substitutions. If P is a proposition, P is said to satisfy constraint $C = \langle\phi_1, \phi_2\rangle$ if and only if $\phi_1(P) \equiv \phi_2(P)$. P is said to satisfy the set C^s of constraints if and only if it satisfies all constraints in this set. For a constraint set C and a substitution σ, we define $C \circ \sigma = \{\langle\phi_1 \circ \sigma, \phi_2 \circ \sigma\rangle \mid \langle\phi_1, \phi_2\rangle \in C\}$.

For instance, let $\langle\phi_1, \phi_2\rangle$ be a constraint with $\phi_1 = id$ and $\phi_2 = \{x \mapsto y\}$. Proposition $P(x)$ does not satisfy this constraint, since $\phi_1(P(x)) = P(x)$, whereas $\phi_2(P(x)) = P(y)$. However, $P(z)$ does meet the constraint, since $\phi_1(P(z)) = \phi_2(P(z)) = P(z)$.

Lemma 75 (Inference of specification constraints without fake)
Let Γ be a Hoare context and S be a program without **fake** *statements. We can compute a triple $T = \langle\phi, \mathcal{Q}^s, C^s\rangle$ where ϕ is a substitution, $\mathcal{Q}^s$ is a set of propositional patterns and C^s is a set of constraints, such that $\Gamma \vdash S{:}P \rhd Q$ if and only if there exists a P' satisfying C^s such that $Q \in \mathcal{Q}^s(P')$ and $P \equiv \phi(P')$.*

Hence, for each P' satisfying C^s we have for each propositional pattern $\mathcal{Q} \in \mathcal{Q}^s$ that

$$\Gamma \vdash S{:}\phi(P') \triangleright \mathcal{Q}(P').$$

Intuitively, P' satisfying C^s is the piece of the postcondition that corresponds to the precondition. The remaining part of the postcondition, introduced by *If*-guards and blocks, is given by the patterns stored in $\mathcal{Q}^s$.

Proof: Let $\Gamma \equiv \langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle$ denote a Hoare context. We use induction on the structure of S (Note that $P_1 \wedge P_2 \wedge \ldots \wedge P_n$ stands for $((P_1 \wedge P_2) \wedge \ldots \wedge P_n)$ and not $P_1 \wedge (P_2 \wedge \ldots \wedge P_n)$ etc.):

case $S \equiv$ **skip:** Use $T = \langle id, \{\alpha\}, \emptyset \rangle$.

case $S \equiv x := e$**:** Use $T = \langle \{x \mapsto e\}, \{\alpha\}, \emptyset \rangle$.

case $S \equiv$ **if** g **then** S_1 **else** S_2 **fi:** By induction we compute for S_1 and S_2 the triples $T_1 = \langle \phi_1, \mathcal{Q}_1^s, C_1^s \rangle$ and $T_2 = \langle \phi_2, \mathcal{Q}_2^s, C_2^s \rangle$ respectively. We choose $T = \langle \phi_1, \mathcal{Q}^s, C^s \rangle$, where $\mathcal{Q}^s = \bigcup_{g' \in G} (\mathcal{Q}_1^s \cap \mathcal{Q}_2^s)(g')$ with $G = \{g' \in \phi_1^{-1}(g = \textbf{true}) \cap \phi_2^{-1}(g = \textbf{false}) \mid g' \text{ satisfies } C_1^s \cup C_2^s\}$ and $C^s = C_1^s \cup C_2^s \cup \{\langle \phi_1, \phi_2 \rangle\}$. Since this is not obvious, we will provide further proof:

Suppose $\Gamma \vdash S{:}P \triangleright Q$. Hence, from the premises of *If* in the Hoare logic we get $\Gamma \vdash S_1{:}P \wedge g = \textbf{true} \triangleright Q$ and $\Gamma \vdash S_2{:}P \wedge g = \textbf{false} \triangleright Q$. Hence, by induction there exist P_1 and P_2 satisfying C_1^s and C_2^s respectively, such that $P \wedge g = \textbf{true} \equiv \phi_1(P_1)$ and $P \wedge g = \textbf{false} \equiv \phi_2(P_2)$ and $Q \in \mathcal{Q}_1^s(P_1)$ and $Q \in \mathcal{Q}_2^s(P_2)$. Hence, P_1 is of the form $P_1' \wedge G_1$ and P_2 is of the form $P_2' \wedge G_2$, such that $P \equiv \phi_1(P_1')$, $P \equiv \phi_2(P_2')$, $g = \textbf{true} \equiv \phi_1(G_1)$ and $g = \textbf{false} \equiv \phi_2(G_2)$. Hence, Q is of the form $P_1' \wedge G_1 \wedge A_1 \wedge \ldots \wedge A_n$ and of the form $P_2' \wedge G_2 \wedge B_1 \wedge \ldots \wedge B_m$. It follows that $m = n$, $P_1' \equiv P_2'$, $G_1 \equiv G_2$ and for all i, $1 \le i \le n$ $A_i \equiv B_i$. Hence, $\alpha \wedge A_1 \wedge \ldots \wedge A_n \in \mathcal{Q}_1^s \cap \mathcal{Q}_2^s$ and also $Q \in ((\mathcal{Q}_1^s \cap \mathcal{Q}_2^s)(\alpha \wedge G_1))(P_1')$. $G_1 \equiv G_2$ satisfies both C_1^s and C_2^s. Since $g = \textbf{true} \equiv \phi_1(G_1)$ and $g = \textbf{false} \equiv \phi_2(G_1)$, we have $G_1 \in \phi_1^{-1}(g = \textbf{true}) \cap \phi_2^{-1}(g = \textbf{false})$, hence $Q \in \bigcup_{g \in G} (\mathcal{Q}_1^s \cap \mathcal{Q}_2^s)(\alpha \wedge G_1)(P_1')$. Also, $P_1' \equiv P_2'$, so P_1' satisfies C_1^s and C_2^s and $\phi_1(P_1') \equiv \phi_2(P_1')$, hence P_1' satisfies $C_1^s \cup C_2^s \cup \{\langle \phi_1, \phi_2 \rangle\}$. If follows that there exists a P' satisfying C^s such that $P \equiv \phi_1(P')$ and $Q \in \mathcal{Q}^s(P')$.

Conversely, suppose there exists a P' satisfying C^s. Let Q be an element of $\mathcal{Q}^s(P')$, i.e. Q has the form $P' \wedge G \wedge A_1 \wedge \ldots \wedge A_n$, where $G \in \phi_1^{-1}(g = \textbf{true}) \cup \phi_2^{-1}(g = \textbf{false})$ and $\alpha \wedge A_1 \wedge \ldots \wedge A_n$ is an element of both $\mathcal{Q}_1^s$ and $\mathcal{Q}_2^s$, and where G satisfies C_1^s and C_2^s. Hence,

$P' \wedge G$ satisfies C_1^s and C_2^s and hence, by the induction hypothesis, $\Gamma \vdash S_1{:}\phi_1(P' \wedge G) \rhd Q$ and $\Gamma \vdash S_2{:}\phi_2(P' \wedge G) \rhd Q$. This is equal to $\Gamma \vdash S_1{:}\phi_1(P') \wedge g = \mathbf{true} \rhd Q$ and $\Gamma \vdash S_2{:}\phi_2(P') \wedge g = \mathbf{false} \rhd Q$. Since P' satisfies $\langle \phi_1, \phi_2 \rangle$ we can now derive $\Gamma \vdash S{:}\phi_1(P) \rhd Q$ for any $Q \in \mathcal{Q}^s(P')$.

case $S \equiv$ **while** g **do** S_1 **od:** By induction, compute $T_1 = \langle \phi_1, \mathcal{Q}_1^s, C_1^s \rangle$ for S_1. Now suppose $\Gamma \vdash S{:}P \rhd Q$ for certain P and Q. From the Hoare logic we get that $Q \equiv P \wedge g = \mathbf{false}$ and that $\Gamma \vdash S_1{:}P \wedge g = \mathbf{true} \rhd P$. Hence, there exists a P_1, satisfying C_1^s such that $P \wedge g = \mathbf{true} \equiv \phi_1(P_1)$ and $P \in \mathcal{Q}_1^s(P_1)$. But if $P \wedge g = \mathbf{true} \equiv \phi_1(P_1)$, then P_1 must have the form $P_1' \wedge G$ such that $P \equiv \phi_1(P_1')$ and $g = \mathbf{true} \equiv \phi_1(G)$. Hence, $P \in \mathcal{Q}_1^s(P_1' \wedge G)$ and $P \equiv \phi(P_1')$, which is impossible, since P is of the form $(\ldots((P_1' \wedge G) \wedge A_1) \wedge \ldots \wedge A_n)$. Therefore, in a program without **fake** statements, **while** statements do not occur.

case $S \equiv |[\mathbf{var}\ x := e : U \ \bullet\ S_1]|$**:** By induction, compute $T_1 = \langle \phi_1, \mathcal{Q}_1^s, C_1^s \rangle$ for S_1. Use $T = \langle \phi_1, \mathcal{Q}^s, C^s \rangle$, where $\mathcal{Q}^s = \{Q(X) \mid X \in \phi_1^{-1}(x = e), X$ satisfies $C_1^s, Q \in \mathcal{Q}_1^s, x \notin FV(Q(X))\}$ and $C^s = C_1^s \cup \{\langle id, \{x \mapsto y\}\rangle, \langle \phi_1, \{x \mapsto y\} \circ \phi_1 \rangle\}$ for some arbitrary $y \not\equiv x$.

Suppose $\Gamma \vdash S{:}P \rhd Q$ for certain P and Q. From the Hoare logic we get that $\langle \Gamma_1, (\Gamma_2, x : U), \Gamma_3 \rangle \vdash S_1{:}P \wedge x = e \rhd Q$ and $x \notin FV(P, Q)$. Hence, there exists a P_1 satisfying C_1^s such that $Q \in \mathcal{Q}_1^s(P_1)$ and $P \wedge x = e \equiv \phi_1(P_1)$. Hence, P_1 has the form $P_1' \wedge X$ with $P \equiv \phi_1(P_1')$ and $x = e \equiv \phi_1(X)$ with P_1' and X satisfying C_1^s. Hence, Q has the form $P_1' \wedge X \wedge A_1 \wedge \ldots \wedge A_n$. Since $x \notin FV(P, Q)$, we know $x \notin FV(P_1', X, A_1, \ldots, A_n)$, hence $P_1' \equiv \{x \mapsto y\}(P_1')$ and $\phi_1(P_1') \equiv (\{x \mapsto y\} \circ \phi_1)(P_1')$. Hence, P_1' satisfies C^s. Also, since $x = e \equiv \phi_1(X)$, $x \notin FV(A_1, \ldots, A_n)$ and $\alpha \wedge A_1 \wedge \ldots \wedge A_n \in \mathcal{Q}_1^s$, we get $Q \in \mathcal{Q}^s(P_1')$.

Conversely, suppose P' satisfies C^s. Let Q be an element of $\mathcal{Q}^s(P')$. Then Q has the form $P' \wedge X \wedge A_1 \wedge \ldots \wedge A_n$, with $x \notin FV(X, A_1, \ldots A_n)$. Also, since P' satisfies $\langle id, \{x \mapsto y\}\rangle$, $x \notin FV(P')$. Hence, $x \notin FV(Q)$. Also, since P' satisfies C_1^s and X satisfies C_1^s, $P' \wedge X$ satisfies C_1^s. By T_1 we get $\langle \Gamma_1, (\Gamma_2, x : U), \Gamma_3 \rangle \vdash S_1{:}\phi_1(P' \wedge X) \rhd Q$. Since P' satisfies $\langle \phi_1, \{x \mapsto y\} \circ \phi_1 \rangle$, $x \notin FV(\phi_1(P'))$. Hence, since $x = e \equiv \phi_1(X)$, we can use *Block* to derive $\Gamma \vdash S{:}\phi_1(P') \rhd Q$.

case $S \equiv S_1; S_2$**:** By induction compute $T_1 = \langle \phi_1, \mathcal{Q}_1^s, C_1^s \rangle$ for S_1 and $T_2 = \langle \phi_2, \mathcal{Q}_2^s, C_2^s \rangle$ for S_2. Use $T = \langle \phi_1 \circ \phi_2, \mathcal{Q}^s, C^s \rangle$, where $\mathcal{Q}^s = \mathcal{Q}_2^s(\{Q \mid \phi_2(Q) \in \mathcal{Q}_1^s, Q$ satisfies $C_2^s\})$ and $C^s = C_1^s \circ \phi_2 \cup C_2^s$. This requires further proof:

Suppose $\Gamma \vdash S{:}P \rhd Q$. From the premises of *Comp* we get $\Gamma \vdash S_1{:}P \rhd R$

and $\Gamma \vdash S_2{:}R \triangleright Q$ for certain R. Hence, there exists a P_2 satisfying C_2^s such that $Q \in \mathcal{Q}_2^s(P_2)$ and $R \equiv \phi_2(P_2)$ and a P_1 satisfying C_1^s such that $R \in \mathcal{Q}_1^s(P_1)$ and $P \equiv \phi_1(P_1)$. Hence, R has the form $P_1 \wedge A_1 \wedge \ldots \wedge A_n$ and since $R \equiv \phi_2(P_2)$, P_2 must have the form $P_1' \wedge A_1' \wedge \ldots \wedge A_n'$ with $P_1 \equiv \phi_2(P_1')$, $A_i \equiv \phi_2(A_i')$ and P_1' and A_i' satisfying C_2^s. Since P_1 satisfies C_1^s, we have that P_1' satisfies $C_1^s \circ \phi_2$, hence P_1' satisfies $C_1^s \circ \phi_2 \cup C_2^s$. Also $P \equiv \phi_1(P_1) \equiv \phi_1(\phi_2(P_1'))$. It remains to show that $Q \in \mathcal{Q}_2^s(\{Q \mid \phi_2(Q) \in \mathcal{Q}_1^s, Q \text{ satisfies } C_2^s\})$. Since P_2 has the form $P_1' \wedge A_1' \wedge \ldots A_n'$ with $A_i \equiv \phi_2(A_i')$ and A_i' satisfying C_2^s, $P_2 \in \{Q \mid \phi_2(Q) \in \mathcal{Q}_1^s, Q \text{ satisfies } C_2^s\}$. From $Q \in \mathcal{Q}_2^s(P_2)$ we get $Q \in \mathcal{Q}_2^s(\{Q \mid \phi_2(Q) \in \mathcal{Q}_1^s, Q \text{ satisfies } C_2^s\})$. Hence, P_1' satisfies C^s, $P \equiv \phi_1 \circ \phi_2(P_1')$ and $Q \in \mathcal{Q}^s(P_1')$.

Conversely, suppose P' satisfies $C_1^s \circ \phi_2 \cup C_2^s$. Let $Q \in \mathcal{Q}^s(P')$. Hence, P' satisfies C_2^s and Q has the form $P' \wedge A_1' \wedge \ldots \wedge A_n' \wedge B_1 \wedge \ldots \wedge B_m$ with A_i' satisfying C_2^s. From T_2 we get $\Gamma \vdash S_2{:}\phi_2(P' \wedge A_1' \wedge \ldots \wedge A_n') \triangleright Q$, i.e. $\Gamma \vdash S_2{:}\phi_2(P') \wedge A_1 \wedge \ldots \wedge A_n \triangleright Q$ with $\alpha \wedge A_1 \wedge \ldots \wedge A_n \in \mathcal{Q}_1^s$. Since P' satisfies $C_1^s \circ \phi_2$, we know $\phi_2(P)$ satisfies C_1^s. Hence, by induction we get $\Gamma \vdash S_1{:}\phi_1 \circ \phi_2(P') \triangleright \phi_2(P') \wedge A_1 \wedge \ldots \wedge A_n$. Using *Comp* we derive $\Gamma \vdash S{:}\phi_1 \circ \phi_2(P) \triangleright Q$.

Since $Q \in \mathcal{Q}_2^s(P_2)$, it has the form $P_2 \wedge B_1 \wedge \ldots \wedge B_m$, which has the form $P_1' \wedge A_1' \wedge \ldots \wedge A_n' \wedge B_1 \wedge \ldots \wedge B_m$. with $\alpha \wedge \phi_2(A_1') \wedge \ldots \wedge \phi_2(A_n') \in \mathcal{Q}_1^s$. Also, for i, $1 \leq i \leq n$ we have that A_i' satisfies C_2^s and $A_i' \in \phi_2^{-1}(A_i)$. Hence, Q has the form $P_1' \wedge A_1' \wedge \ldots \wedge A_n' \in \mathcal{Q}_1^s$

case $S \equiv$ **fake** p**:** Since S contains no **fake** statements, this case cannot occur.

□

As an example, consider the following simple program[2](in this program, g and b are variables of type boolean):

```
S ≡ if g then b := true
          else b := false
    fi
```

For the assignments $b :=$ **true** and $b :=$ **false**, the triples $T_1 = \langle\{b \mapsto \textbf{true}\}, \{\alpha\}, \emptyset\rangle$ and $T_2 = \langle\{b \mapsto \textbf{false}\}, \{\alpha\}, \emptyset\rangle$ are computed. Then, for

[2]Programs without **fake** statements for which valid specifications exist will always be somewhat artificial. However, for the sake of completeness of the theory, these programs must also be considered.

the **if**-statement, the constraint set becomes $\emptyset \cup \emptyset \cup \{\langle\{b \mapsto \mathbf{true}\}, \{b \mapsto \mathbf{false}\}\rangle\}$. To compute the propositional patterns, we first compute the inverse substitutions $\{e \mapsto \mathbf{true}\}^{-1}(g = \mathbf{true}) = \{g = \mathbf{true}, g = b\}$ and $\{e \mapsto \mathbf{false}\}^{-1}(g = \mathbf{false}) = \{g = \mathbf{false}, g = b\}$. The intersection of these sets is $\{g = b\}$, which trivially satisfies the empty sets of constraints. Hence, the triple for the **if**-statement becomes $T = \langle\{b \mapsto \mathbf{true}\}, \{\alpha \wedge g = b\}, \{\langle\{b \mapsto \mathbf{true}\}, \{b \mapsto \mathbf{false}\}\}\rangle$.

To check the result, consider the proposition $a = \mathbf{true}$. It meets all constraints, hence $S : a = \mathbf{true} \triangleright a = \mathbf{true} \wedge g = b$ must hold, which is easy to derive from the *If* and *Assign* rules of *While*.

Conversely, the proposition $a = b$ does not meet the constraint and hence, $S : a = \mathbf{true} \triangleright a = b \wedge g = b$ should not hold. Indeed, the *Assign*-rule of *While* states $b := \mathbf{true} : a = \mathbf{true} \wedge g = \mathbf{true} \triangleright a = b \wedge g = b$ and $b := \mathbf{false} : a = \mathbf{false} \wedge b = \mathbf{false} \triangleright a = b \wedge g = b$. Hence, the *If*-rule of *While* cannot be applied.

Corollary 76 (Specification inference is decidable)
From lemma 72 and lemma 75 it immediately follows that specification inference is decidable. (For programs without **fake**, *the proposition* $\mathbf{true} = \mathbf{true}$ *will always meet all constraints).*

Chapter 11

Results

We now have a formal system combining interactive theorem proving with both automated theorem proving and program derivation. Moreover, the entire system is based on the semantics of first-order logic described in chapter 5. Also, the correctness of both proofs and programs can be verified, even if parts of the proof were generated automatically.

$\lambda P-$ was designed to accurately describe first-order logic in a PTS, hence enabling the combination of interactive and automated theorem proving. Yet, since it is a PTS, proofs can be communicated to other PTS-based systems. Even the additional features are easily converted, provided that the target PTS is powerful enough to express the features axiomatically.

Translating tableaux into λ-terms showed some advantages: the automated theorem prover can be extended without extending the logic unexpectedly, provided that its result will remain an ordinary closed tableau. However, to incorporate Leibniz-equality, more work needs to be done. Although tableau methods dealing with equality are known (see [dK95, DGHP99, BS98]), we have not yet investigated how these tableaux can be translated into λ-terms.

The Hoare logic was designed to have properties similar to those of a PTS, which eased the combination of the two formalisms. Also, this enabled us to use a simple logic, rather than the higher order logics required to embed the entire Hoare logic. However, it is desirable to extend the Hoare logic with more advanced features like records, sub-typing, classes and pointers. Some of these features require an extension of the logic and hence, of $\lambda P-$. For instance, in [Zwa99] Jan Zwanenburg discussed PTSs with records and sub-typing. Richard Bornat is currently using JAPE to verify pointer semantics for Hoare Logic based on an idea of Rod Burstal (see [Bor00, Bur72]).

However, since the goal of this thesis is to create an educational tool as a proof of concept, we will not discuss further extensions of the formal basis of the system. Instead, we will focus on the construction of a tool for the presented formalism in the next part.

Part III

System Design

Chapter 12

Introduction

This part of the thesis describes the actual design and implementation of Cocktail. If one wants to implement a tool to support the formalisms described in part II, one must realize that there is a large gap between the formalisms and a useful system. The fact that the formalism is sound and complete does not imply that it reflects the way programmers think about their programs. The formalism merely embodies the axioms that are used by the system.

In practice, systems are built to construct and keep account of a whole set of theorems, as opposed to the formalism which, in general, deals with a single theorem. Cocktail uses the same approach to building sets of theorems as Yarrow, COQ and LEGO: Instead of re-constructing a context for each theorem to be proved, it uses a single context that is globally available throughout the system. Whenever a theorem has been proved within this global context, the context will be extended to include it. That is, if the derivation $\Gamma \vdash A{:}B$ is completed, the global context Γ is extended with a definition $H = A : B$, where H is the name of the theorem. During the construction of any other theorem, H can be used where necessary.

12.1 Design considerations

Programmers will often use several rules in the same order to achieve a certain goal. These series of rule-applications are called tactics. Instead of providing the rules of the formalism directly to the user, it is recommended to provide useful tactics that reflect the way programmers think about program derivation (or proof construction). An even better approach is to allow the programmer to construct such tactics herself from the rules of the formalism.

Another issue is the presentation of the data to the user. Certainly, displaying the λ-term or program under construction is not sufficient: special attention is needed for the lay-out of formulas, proofs and programs. Simply presenting a huge bulk of text representing context, proof and theorems does not look very attractive to a potential user. Moreover, the user will not be able to overview the problem at hand and hence, might be unable to construct simple proofs. Also notations must appear as natural as possible: having a user deciphering every formula displayed will cost much energy that is better spent on constructing proofs and programs. Hence, what information should be displayed and how it should be displayed is a major issue if the system is to be used by users other than the system's designer.

However, even a system providing a good symbolic engine, useful tactics and easy to read formulas and proofs will not be used if users are barely able to interact with the system. Hence, an intuitive user-interface is needed, which enables the user to apply the desired tactic in a simple way. The design of the actual user interface will be based on the way in which formulas and proofs are displayed.

12.2 Implementation issues

The implementation of Cocktail was done incrementally. First the symbolic engine was constructed, which consists of data structures to represent formulas, proofs and programs and an algorithm to check the correctness of these data structures. This algorithm, known as type-checking, ensures the reliability of the entire system, since the correctness of every proof and program is checked after it has been constructed. Hence, if errors were made by the more complex parts of the tool during the construction of the data representations, these will be detected in the end.

Because of the importance of type-checking, the correctness of the type checker is crucial, which is one of the main reasons for using a small and simple logic. Type checking for $\lambda P-$ is straightforward since it has no conversion rule. The type-checker for programs is syntax-driven, since the conclusions of each of the axioms and rules of *While* has a different syntactic structure.

Once the symbolic engine was finished, editors were implemented to manipulate syntactic structures of the system. These editors enabled the construction of proofs and programs. Since the proof and program editors are able to apply tactics to open parts of the structures based on their type or specification, the level at which the user will construct proofs will surpass the level of the axiomatic systems defined in part II. For instance, instead of only providing a Leibniz-equality rule, it is possible to define groups of rewrite-rules (either assumed axioms or proved theorems) that can be applied to a formula at once. The editor will then repeatedly apply one of these rules to the formula, until no more rules apply (hence, the formula is rewritten into a normal form of the defined system, if it exists).

In order to make Cocktail more applicable and workable, a graphical user interface was designed on top of the symbolic engine and the tactic system. This user interface consists of the display of formulas, proofs and programs and the interaction with the system. The display was designed to be as natural as possible, using notations the users are familiar with (e.g. infix notation for propositional and functional constructs). The user interface utilizes modern interface techniques like simple mouse-clicks, pop-up menus, drag-and-drop of formulas etc.

The implementation is described in more detail in chapter 15. However, there are two issues to be treated that are difficult to place in this thesis, yet are important to the tool: equality for propositions and unification. Equality for propositions is important, since users will want to rewrite propositions as well as expressions, even though Leibniz equality is defined only for the latter. Unification is important for the implementation of the tableau based automated theorem prover (see chapter 7), but also for some of the tactics (see section 15.3). We will discuss both issues here in separate subsections.

12.2.1 Leibniz equality for propositions

In our definition of $\lambda P-$, we added rules to allow Leibniz equality for values of data-types. In practice, however, the user also wants to use equality for propositions, like $A \wedge B = \neg(\neg A \vee \neg B)$. Adding Leibniz for propositions would not extend the logic, which is stated by the following lemma:

Lemma 77 *If $\Gamma \vdash a{:}A \Rightarrow B$ and $\Gamma \vdash b{:}B \Rightarrow A$ for certain propositions A and B in $\lambda P-$, and $\Gamma \vdash p{:}Q[\odot := A]$ for certain proposition $Q[\odot := A]$, then there exists a p' such that $\Gamma \vdash p'{:}Q[\odot := B]$.*

Proof: By induction on the structure of Q. $\square$

The premises of this lemma are similar to the premises of the Leibniz rule of $\lambda P-$ on page 64, except that it requires $A \Rightarrow B$ and $B \Rightarrow A$ to be proved rather than $A = B$. Hence, adding Leibniz equality for propositions does not extend the logic ($A \Rightarrow A$ is trivial to prove for any proposition A, hence reflexivity is also valid). This is not done for two reasons:

1. $\lambda P-$ should remain as small as possible, hence adding rules that do not extend the logic is avoided.
2. Logicians would object to the identification of $A \Rightarrow B$ and $B \Rightarrow A$ with $A = B$. It should read $A \Leftrightarrow B$ instead. In logic, however, Leibniz equality is usually not defined for $\Leftrightarrow$ but applied in theorems at the meta-level of the logic.

Cocktail could transparently allow rewriting, constructing proof-terms as prescribed by the proof of the lemma (not given in detail). This, however, would result in large proof structures, slowing the system down unnecessarily. Instead, the *implementation* internally uses the additional rule

$$= \mathit{intro} \quad \frac{\begin{array}{c}\Gamma \vdash a{:}A \Rightarrow B \\ \Gamma \vdash b{:}B \Rightarrow A\end{array}}{\Gamma \vdash \mathit{in\_equ}\ a\ b{:}A = B}$$

Similarly, a rule to incorporate tableau-based proofs in λ-terms without translating is used:

$$\textit{tab-intro} \quad \frac{\text{"a tableau for } A \text{ was found"}}{\Gamma \vdash \textit{tab } A{:}A}$$

Note that both rules can be eliminated by computing the proof terms for *in_equ a b* and *tab A* respectively. Therefore, these rules are considered to be part of the *implementation* and not of the *formalism.*

12.2.2 Unification

As discussed in chapter 7, a unification algorithm computes the most general unifier for two expressions e_1 and e_2. That is, it computes a substitution θ, such that $\theta(e_1) = \theta(e_2)$. Computing most general unifiers is needed, for instance, to decide whether a tableau-leaf is open or closed or to derive new clauses in a resolution proof. In interactive theorem proving, unification is used to automatically compute arguments for universally quantified formulas, such that the instantiation of this formula matches with a given goal. An algorithm to compute most general unifiers for first-order formulas was first described by Robinson in [Rob65].

In [MM82], Martelli and Montanari describe an efficient unification algorithm. Their algorithm, however, requires somewhat complicated data-structures and requires the tree-representation of formulas to be inspected in a specific order. In this section, we will show how an efficient unification algorithm can be constructed that uses no complicated data-structures and that does not require the tree-representation of formulas to be inspected in any specific order. Unfortunately, we have not yet been able to compare computing times of our algorithm with computing times of Martelli and Montanari's algorithm.

We start with a basic unification algorithm that can be formulated as follows (see [NN92]).

$\theta := id; S := \{e_1 = e_2\}$

1. If $S = \emptyset$ we are done: θ is a most general unifier.
2. Select an equation from S, say $s = t$, and remove it from S.
3. If s is a variable then

   (a) if t is the same variable, then go to step 1.
   (b) else if s occurs in t, then abort with failure (occurs check).
   (c) else replace s by t in S and replace θ by $[s \mapsto t] \circ \theta$.

4. If s has the form $fs_1 \ldots s_m$ and t is a variable then add $t = s$ to S.

5. If s and t have the forms $fs_1 \ldots s_m$ and $gt_1 \ldots t_n$ respectively, then
   (a) if $f = g$ add $s_1 = t_1$ till $s_m = t_n$ to S (since $f = g$ we know $m = n$).
   (b) else abort with failure (function mismatch).

6. Return to step 1.

Correctness and termination of this algorithm follow from the observation that the set of most general unifiers of S does never change under application of the algorithm and that the overall complexity of the formulas in S decreases. A proof can be found in [NN92].

We will show how efficiency can be increased by choosing the correct representations for S and θ.

Note that every substitution in S is followed by the incorporation of the corresponding mapping in θ. Hence, instead of S, we can maintain a set S' of equations and maintain invariant that $\theta(S') = S$. We then get the following version of the unification algorithm:

$\theta := id; S' := \{e_1 = e_2\}$

1. If $S' = \emptyset$ we are done: θ is a most general unifier.

2. Select an equation from S', say $s = t$, and remove it from S'.

3. If $\theta(s)$ is a variable then
   (a) if $\theta(t)$ is the same variable, then go to step 1.
   (b) else if $\theta(s)$ occurs in $\theta(t)$, then abort with failure (occurs check).
   (c) else replace θ by $[\theta(s) \mapsto \theta(t)] \circ \theta$.

4. If $\theta(s)$ has the form $fs_1 \ldots s_m$ and $\theta(t)$ is a variable then add $t = s$ to S'.

5. If $\theta(s)$ and $\theta(t)$ have the forms $fs_1 \ldots s_m$ and $gt_1 \ldots t_n$ respectively, then
   (a) if $f = g$ add $s_1 = t_1$ till $s_m = t_n$ to S' (since $f = g$ we know $m = n$).
   (b) else abort with failure (function mismatch).
6. Return to step 1.

The substitution θ can be represented as a list

$$\theta' = < [x_1 \mapsto t_1], \ldots, [x_n \mapsto t_n] >$$

of one-point mappings, if we maintain invariant that

$$\theta = [x_1 \mapsto t_1] \circ \ldots \circ [x_n \mapsto t_n]$$

The representation for $[x \mapsto \theta(t)] \circ \theta$ is then computed by prepending $[x \mapsto \theta(t)]$ to θ'. But this still requires the computation of $\theta(t)$. However, since this is the only modification we ever make to θ', we choose to represent θ' in turn by another list of one-point substitutions θ'', and maintain the invariant

$$\theta' = < [x_1 \mapsto \theta_1(t_1)], \ldots, [x_n \mapsto \theta_n(t_n)] >$$

where

$$\theta'' = < [x_1 \mapsto t_1], \ldots, [x_n \mapsto t_n] >$$

and

$$\begin{aligned} &\theta_i = [x_{i+1} \mapsto \theta_{i+1}(t_{i+1})] \circ \ldots \circ [x_n \mapsto \theta_n(t_n)] \\ &\theta_n = id \end{aligned}$$

Note that $\theta = \theta_0$ and hence, θ can be computed directly from θ'' and we no longer need θ' at all. Prepending $[x \mapsto \theta(t)]$ to θ' is then equal to prepending $[x \mapsto t]$ to θ''.

During the computation of the unifier, it is necessary to compare the first symbols of terms, i.e. the variable if the terms are variables and the function symbol if the terms are function applications. Whatever the representation of our unifier θ is, we must be able to compute the first symbol of $\theta(s)$ for a term s. Therefore, we use the following lemma.

Lemma 78 *Let $\theta'' = < [x_1 \mapsto t_1], \ldots, [x_n \mapsto t_n] >$ be a list of one-point mappings and let θ be the corresponding substitution as defined above. Also assume that x_i does not occur in $\theta_i(t_i)$ for any i, and that all x_i are different. Let v be a variable, such that $v = x_i$ for certain i. Then $\theta(v) = \theta(t_i)$.*

Proof:

$$\begin{array}{cl}
& \theta(v) \\
= & \{\text{definition of } \theta\} \\
& ([x_1 \mapsto \theta_1(t_1)] \circ \ldots \circ [x_n \mapsto \theta_n(t_n)])(v) \\
= & \{v \notin \{x_{i+1}, \ldots, x_n\}\} \\
& ([x_1 \mapsto \theta_1(t_1)] \circ \ldots \circ [x_i \mapsto \theta_i(t_i)])(v) \\
= & \{v = x_i;\text{definition of } \circ\} \\
& ([x_1 \mapsto \theta_1(t_1)] \circ \ldots \circ [x_{i-1} \mapsto \theta_{i-1}(t_{i-1})])(\theta_i(t_i)) \\
= & \{x_i \text{ does not occur in } \theta_i(t_i)\} \\
& ([x_1 \mapsto \theta_1(t_1)] \circ \ldots \circ [x_i \mapsto \theta_i(t_i)])(\theta_i(t_i)) \\
= & \{\text{definition of } \circ \text{ and } \theta_i\} \\
& \theta(t_i)
\end{array}$$

□

This lemma allows us to insert steps 2(a) and 2(b) into the algorithm. These steps will compute the image $\theta(s)$ and $\theta(t)$ until a non-variable is encountered or until the full image is computed. This will eliminate most terms of the form $\theta(x)$ in the algorithm. The algorithm then becomes:

$\theta'' :=<>; S' := \{e_1 = e_2\}$

1. If $S' = \emptyset$ we are done: θ'' represents a most general unifier.
2. Select an equation from S', say $s = t$, and remove it from S'.
   (a) While $s = x_i$ for some x_i occuring in θ'', replace s by the corresponding t_i.
   (b) While $t = x_i$ for some x_i occuring in θ'', replace t by the corresponding t_i.
3. If s is a variable then
   (a) if t is the same variable, then go to step 1.
   (b) else if s occurs in $\theta(t)$, then abort with failure (occurs check).
   (c) else prepend $[s \mapsto t]$ to θ''.
4. If s has the form $f s_1 \ldots s_m$ and t is a variable then add $t = s$ to S'.
5. If s and t have the forms $f s_1 \ldots s_m$ and $g t_1 \ldots t_n$ respectively, then
   (a) if $f = g$ add $s_1 = t_1$ till $s_m = t_n$ to S' (since $f = g$ we know $m = n$).

   (b) else abort with failure (function mismatch).

6. Return to step 1.

The only occurrence of θ in this version is in step 3(b), where we need to compute if s occurs in $\theta(t)$. Since we do not want a direct representation of θ, but we already have a representation of θ'', we will check for occurrences of s in $\theta(t)$ with the following algorithm:

$V := \{t\}$

1. If $V = \emptyset$ we are done: s does not occur in t.
2. Select a term from V, say t' and remove it from V.
3. While $t' = x_i$ for some x_i occurring in θ'', replace t' by the corresponding t_i.
4. If t' is the same variable as s, we are done: s does occur in t.
5. If t' has the form $gt_1 \dots t_n$ then add t_1 till t_n to V.
6. Return to step 1.

Hence, our final algorithm does not impose any order in which formulas of the set of equations have to be chosen. Also, the only required data-structure is a list of pairs, in which the first element of each pair refers to a variable and the second element refers to the subtree that should be substituted for this variable.

Chapter 13

Design Goals

This chapter discusses a number of design issues we want to keep in mind when designing Cocktail. These issues, however, are not specific for Cocktail: they should be considered for every system one designs. We address the issues of modularity, flexibility and usability in separate sections.

13.1 Modularity

The main reasons for a modular design are the transparency and maintainability of the code. By keeping concepts apart and implementing these in separate chunks, it will be easier to find and change all pieces of code related to the concepts. For instance, the way in which formulas are drawn on the screen (rendered) should not depend on code scattered all over the system.

Instead, all parts used to render formulas should be kept in a limited number of classes, packages or components.

Modularity affects the system at all levels: the symbolic engine can be set apart from the user-interface. Within the symbolic engine, the type-checker can be set apart from the term representation. Then, within the type-checker, rules for regular CPTSs can be set apart from rules specific for $\lambda P-$ etc.

Modularity is supported by virtually all programming languages. C uses headers and libraries, C++ adds object to this, Turbo-Pascal uses units and Object Pascal (Delphi) combines units and objects. Java has packages, being collections of objects. Hence, the modularity of the design is not dependent on the language used for implementation.

Cocktail has been designed modularly at many levels. For instance, the formalisms for CPTSs in general, $\lambda P-$, the tableau based theorem prover and the Hoare logic are kept in separate packages. Also, the user-interface is kept apart from the editors, that do the actual manipulation of proof terms. Within the user-interface, the display routines are implemented apart in only a few classes. Modularity is also established by using some design patterns (see [GHJV95]).

13.2 Flexibility

The tool should be flexible in the sense that it must allow the user to work in her own way. One must be able to define functions in one's own preferred way, use notations one is used to and perform reasoning steps in a natural order.

Also, the tool should be reasonably adjustable to other logics, although it will be necessary to change the program to change the logic it uses. If we would make the logic a parameter of the system, the tool would become generic which can be considered to be the ultimate state of flexibility. However, this is not done for two reasons:

- Not all logics will be suitable, which becomes apparent if one considers the requirements needed to link the Hoare logic to the theorem prover. Implementing the combination of program-derivation and theorem prover would become more difficult in a generic system (see also chapter 10).

- Generic systems are, in general, usually applied to only one logic. For instance, Isabelle is used with the HOL logic most of the time (see [Pau94]). JAPE comes with only one logic that really uses the capabilities of the generic tool (see [BS]). Most other logics supported by such tools are more used as proofs of concept than as actual systems themselves. We rather concentrate on building a tool using a single logic than trying to build a generic system which, because of time constraints, could only be tested for one logic. This makes our job considerably easier and gives hope to obtain more results in the same amount of time. It also enables more specific support for the logic being used.

Flexibility is made easier by the modular approach: to provide a flexible, user definable notation, one only has to change the module responsible for displaying terms. To allow new ways of reasoning, only the proof-editor has to be adjusted instead of the entire system.

13.3 Usability

Flexibility is necessary to create a usable tool, but it is not sufficient. In order for the tool to become usable, it must provide many additional features, enabling the user to use the tool with a minimum of fuss. These features include, but are not limited to:

1. cut, copy and paste functionality.
2. an embedded automated theorem prover
3. extensive help functions and documentation.
4. a useful, extensible tactic system.
5. a well-organized workspace.
6. the possibility to reorganize the information, notably the global context.
7. load and save functionality, if possible for parts of context as well as the full workspace.

8. theory browsing facilities.

Although Cocktail aims to become a useful system, not all of these features have been implemented yet (notably, the features 3,6,8). Technically, these features are not hard to realize, but it would take a considerable amount of time, which was not available at the time of this writing. However, many of such features are scheduled for implementation in the near future.

Chapter 14

A Graphical User Interface

To increase the acceptance of the system, it is necessary to provide the user with an easy-to-use user interface. Therefore, this chapter describes the way we want to visualize formulas, proofs and programs (display) and how we want to let the user communicate with the system (interaction).

14.1 Displaying formulas, proofs and programs

In Cocktail, the difference between a formula, a proof and a program is small, since they are all represented by terms of $\lambda P-$ or the Hoare logic for *While*. Nevertheless, in the mind of a user there is a clear distinction between the three: formulas are either expressions resulting in some data type or propositions that need to be proved. Proofs are logical derivations proving propositions. Programs are algorithms describing an executable computation. Hence, it is not sufficient to simply write the terms of the formalism to the screen, since it will scare away most users by its unreadability. Instead, Cocktail provides extended features to display formulas, proofs and programs. We will discuss them in this order.

14.1.1 Displaying formulas

Cocktail uses $\lambda P-$ for its resemblance with first-order logic. Since most users will be familiar with first-order logic, but not necessarily with typed λ-calculi, we want Cocktail to display all formulas as if they were in first-order logic. In practice, the consequences of this decision are:

- Cocktail must display real symbols like $\forall$, $\exists$, $\wedge$ etc. instead of using ASCII representations of those (like forall, exists, and).
- It is possible that equal formulas must be displayed in different ways. For instance, in $\lambda P-$ the term $\Pi H{:}A.\ \perp$ represents both $A \Rightarrow\perp$ and $\neg A$. Which display version is preferred is context dependent. For instance, how did the user enter the formula?
- There must be facilities for the user to specify notations for newly declared functions and predicates. For instance, if the user adds a function $st(x : nat, y : nat) : bool$ to the language context to represent the smaller-than relation, she will probably prefer $x < y$ to be displayed over $st(x, y)$. Also, it should be possible to specify priorities for these functions, to reduce the number of brackets.

The first two consequences are easily implemented: Java, in which Cocktail is implemented, supports Unicode characters and hence, logical symbols can be displayed directly. Information about how propositions must be displayed is stored in the root node of their tree representation during parsing. Hence, propositions are displayed the way they were entered. The third consequence requires more attention and will be discussed in the following paragraph.

Since $\lambda P-$ was specially designed to eliminate the conversion rule of PTSs, all functions and predicates are defined as parametric constants. That is, the definitions will always be of the form $C(x_1 : U_1, \ldots, x_n : U_n) : U$. To support infix notations, we have to let the user specify how to display $C(e_1, \ldots, e_n)$, where e_1 till e_n are expressions of the correct types. Note, that since all e_i

have a type $U : *_s$, they are either of the form $E(y_1, \dots, y_m)$ or they are a variable. Let the display-string for formula $C(e_1, \dots, e_n)$ be denoted by $\mathcal{F}(C(e_1, \dots, e_n))$. To keep displaying feasible, we make two assumptions:

1. Displaying will be compositional, i.e. the display-string of expression $C(e_1, \dots, e_n)$ consists of the strings of $\mathcal{F}(e_1)$ till $\mathcal{F}(e_n)$ and some additional constant strings.

2. The display strings $\mathcal{F}(e_1)$ till $\mathcal{F}(e_n)$ will occur in this order. That is, $\mathcal{F}(e_1)$ is displayed to the left of $\mathcal{F}(e_{i+1})$ for every i with $1 \leq i < n$.

Under these restrictions, it is sufficient to let the user specify $n+1$ strings s_0 through s_n such that

$$\mathcal{F}(C(e_1, \dots, e_n)) = s_0\text{'('}\mathcal{F}(e_1)\text{')'}s_1 \dots s_{n-1}\text{'('}\mathcal{F}(e_n)\text{')'}s_n.$$

The forced syntactic brackets, denoted as '(' and ')' to distinguish them from the usual meta-brackets, that surround all $\mathcal{F}(e_i)$ are required to avoid confusion about the structure of the expression. Otherwise,

$$\mathcal{F}(times(plus(a, b), d))$$

would be equal to

$$\mathcal{F}(plus(a, times(b, d))),$$

for instance $a + b * d$. With the additional brackets, it is displayed as $(a) + ((b) * (d))$.

The number of displayed brackets can drastically be reduced by specifying priorities for each function and predicate. Constants (nullary functions) and variables should always have the highest priority, hence it is best to represent the highest priority by the lowest number 0. All other numbers can then be used for functions and predicates. Brackets will only be displayed around $\mathcal{F}(e_i)$ if e_i's main symbol has lower priority than C has. Hence, if it has a higher associated number.

Now that the user specified how formulas should be displayed, she will also want to use the same notation to enter them. This results in an impossible parsing problem, since we did not impose any restrictions on the strings specified by the user. Hence, $C(x, y : \mathbf{nat}) : \mathbf{nat}$ and $E(x, y : \mathbf{nat}) : \mathbf{bool}$ might both be displayed as $x + y$. Conversely, it is not clear whether $x + y$ denotes $C(x, y)$ or $E(x, y)$. For practical reasons, we want to be able to parse not only small expressions entered by the user, but also larger files. For this, we want to use a recursive-decent parser, restricting our possible solutions. However, parsing is still possible, if we ensure that the type of the first argument, along with the first non-prefix string uniquely determines which parametric constant was used. Therefore, we will impose the following restrictions on the strings a user is allowed to specify for function and predicate symbols.

Suppose a user has specified string s_0 till s_n for $C(x_1 : U_1, \ldots, x_n : U_n) : U$ and strings t_0 till t_m for $E(y_1 : V_1, \ldots, y_m : V_m) : V$, then:

1. s_1 till $s_{(n-1)}$ and t_1 till $t_{(m-1)}$ must be non-empty and all s_i and t_j that are not empty, must be identifiers as specified by the lexical scanner. We will not discuss the definitions of the lexical scanner in detail, but the strings must be recognizable as such.

2. if neither s_0 nor t_0 is empty, they must be different.

3. If s_0 and t_0 are both empty and U_1 and V_1 are equal, s_1 must be different from t_1.

Under these restrictions, parsing an expression is unambiguously done as follows:

- Read the first token. Assume that a priority p is given, such that operators must exceed this priority in order to be parsed. Initially the value will be the highest possible value (i.e. the lowest priority), since we want to parse all operators.

- If the first token is the name of a variable or a constant, determine its type and store it in a variable, say U_L.

- If the first token is an identifier, then by restriction 2, we can already determine which parametric constant is meant. In that case, parsing is only a matter of recursively parsing the sub-expressions and checking whether the other substrings occur at the correct places. Use the constant's priority as the new limit. After parsing, determine the type and store it in U_L.

- If the first token is a bracket, recursively parse the expression within and determine its type and store it in U_L. Skip the closing bracket. The priority to parse the sub-terms is set to the highest possible value again, since the brackets overrule priorities.

- By now, U_L is the type of the expression that was just parsed. If the parser has more input and the next token is an identifier s_1, then, by restriction 1, the parsed expression is apparently the first argument of a parametric constant whose first display string s_0 is empty. By restriction 3, we can determine which parametric constant is instantiated, since for every type U_L and s_1 at most one parametric constant can be found. If the priority of the parametric constant exceeds the given maximum, use recursive calls to parse the remaining arguments, using the priority of the current parametric constant as the new limit. Then return to the beginning of this step. Otherwise, return to the calling procedure.

For example, assume we have parametric constants $times(x, y : \mathbf{nat}) : \mathbf{nat}$ and $plus(x, y : \mathbf{nat}) : \mathbf{nat}$ displayed as $x*y$ and $x+y$ and with priority values 10 and 20 respectively. Now we want to parse $x + y * z$. We set the priority limit to 30. First, x is parsed. Then the $+$ symbol is encountered, which has higher priority than 30 (indicated by a lower value). Therefore, parsing proceeds recursively with $y * z$, but the limit is set to 20. y is successfully parsed and the $*$ symbol is encountered. Again, it has a higher priority than the limit (now 20) and hence, z is recursively parsed. After that, $y * z$ is returned to the recursive call made while parsing $+$. Therefore, the parsed formula is $x + (y * z)$.

But if we parse $x * y + z$, the following happens: x is parsed successfully. Then $*$ is read and hence, an attempt is made to parse $y+z$ recursively with priority 10. y is parsed successfully and then $+$ is read. But this symbol has a lower priority than the given limit and hence, z is not yet parsed, but the parser returns y as the result, yielding a state in which $x * y$ is parsed and $+$ is the next symbol available. Now, z is successfully parsed with a limit of 20, yielding the parsed formula $(x * y) + z$ as desired.

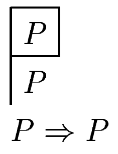
Perhaps that a more complicated parser would allow even more general definitions of the display function $\mathcal{F}$, like different ordering of arguments, non-compositionality etc, but for most purposes the mechanism described above will do.

14.1.2 Displaying proofs

From the display of a proof, the reasoning pattern should become clear. Hence, it should be readable in such a way that the user gets an idea of the

$$\frac{P \vdash P}{\vdash P \Rightarrow P}$$

(a)



(b)

Figure 14.1: A proof of identity, shown as a tree derivation (a) and a natural deduction style proof in flagpole notation (b).

way in which the theorem was proved. Clearly, λ-terms do not have this property when displayed directly.

However, there is a close correspondence between λ-terms and natural deduction style proofs. For instance, a λ-term proving $P \Rightarrow P$ can be $\lambda p : P.p$. The p in the body of the λ-term is actually a repetition of the assumed proof $p : P$ in the binder of the λ-term. In natural deduction, this would be displayed as a tree (figure 14.1a) or in Fitch-style notation (figure 14.1b). The Fitch-style notation (see [Fit52]), also called box-line proofs (e.g. in Jape, see [BS]), has the advantage that it is linear. I.e. only one formula has to be displayed on each line. For this reason, Cocktail will use the latter display. We will define a mapping from λ-terms representing proofs to a display in the next paragraph. The display version used is due to R. Nederpelt (see [Ned77]) and is a more detailed notation than the original Fitch-style display. It has been used many years by students at the Eindhoven University of Technology, and is used in a slightly different way since 1999.

Let $p : P$ be a λ-term representing a proof p of $P : *_p$. Then the display $\mathcal{P}(p)$ for proof p is defined as shown in figure 14.2 ($\mathcal{F}$ is the display function for formulas described in the previous subsection).

Following this definition, we can display the proof

$$(\lambda H{:}A.\ \lambda G{:}B.\ \lambda I{:}A \Rightarrow \neg B.\ I\ H\ G)$$

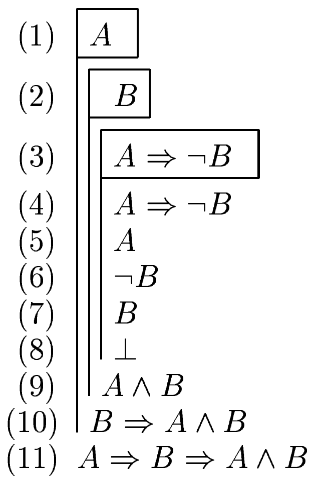
of

$$A \Rightarrow B \Rightarrow A \wedge B$$

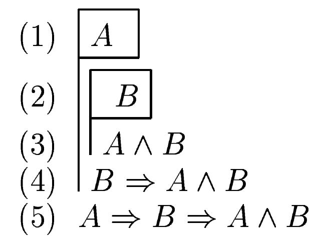
as (the numbers are added for referencing purposes):

$$\begin{array}{llll}
\text{case } p \in V & \mathcal{P}(p) \equiv & \mathcal{F}(P) & \text{if } P : *_p, \\
 & & & \text{nothing otherwise.} \\[1ex]
\text{case } p \equiv Fa & \mathcal{P}(p) \equiv & \begin{array}{l} \mathcal{P}(F) \\ \mathcal{P}(a) \\ \mathcal{F}(P) \end{array} & \\[1ex]
\text{case } p \equiv (\lambda H{:}A.\ b) & \mathcal{P}(p) \equiv & \begin{array}{l} \begin{array}{|l} \boxed{\mathcal{F}(A)} \\ \mathcal{P}(b) \end{array} \\ \mathcal{F}(P) \end{array} & \text{if } A : *_p \\[1ex]
 & & \begin{array}{l} \begin{array}{|l} \boxed{H : A} \\ \mathcal{P}(b) \end{array} \\ \mathcal{F}(P) \end{array} & \text{if } A : *_s \\[1ex]
\text{case } p \equiv \text{classic } A\ b & \mathcal{P}(p) \equiv & \begin{array}{l} \mathcal{P}(b) \\ \mathcal{F}(P) \end{array} & \\[1ex]
\text{case } p \equiv \text{refl } e & \mathcal{P}(p) \equiv & \mathcal{F}(P) & \\[1ex]
\text{case } p \equiv \text{leib } Q\ p\ e & \mathcal{P}(p) \equiv & \begin{array}{l} \mathcal{P}(p) \\ \mathcal{F}(P) \end{array} & \\[1ex]
\text{case } p \equiv \text{nat\_ind } p_0\ p_i & \mathcal{P}(p) \equiv & \begin{array}{l} \mathcal{P}(p_0) \\ \mathcal{P}(p_i) \\ \mathcal{F}(P) \end{array} &
\end{array}$$

Figure 14.2: Displaying λ-terms in the flagpole notation.


(1) A
(2) B
(3) $A \Rightarrow \neg B$
(4) $A \Rightarrow \neg B$
(5) A
(6) $\neg B$
(7) B
(8) $\bot$
(9) $A \wedge B$
(10) $B \Rightarrow A \wedge B$
(11) $A \Rightarrow B \Rightarrow A \wedge B$

Note that the formula in line (9) might also be displayed as $\neg(A \Rightarrow \neg B)$, but this depends on the information stored in the root node of its tree-representation. The proof demonstrates exactly what we want to avoid: it shows the internal encoding of $A \wedge B$ as $\neg(A \Rightarrow \neg B)$ in $\lambda P-$. But if we hide the flag starting in line (3) and its contents on lines (4) till (8), we get:


(1) A
(2) B
(3) $A \wedge B$
(4) $B \Rightarrow A \wedge B$
(5) $A \Rightarrow B \Rightarrow A \wedge B$

which is what we want. To facilitate this selective hiding, we use a mechanism like the one used for flexible displaying of formulas: we store additional information in the tree representation of the proof term to hide (parts of) a proof. For instance, for a λ-construct, we can choose to hide the assumption (the flagpole), the sub-proof (everything under the flagpole) or the result type (the conclusion directly under the flagpole) independently of each other.

In addition to complete proofs, we also need to display proofs under construction. In such proofs, holes may exist (denoted by the syntactic term ?), but they will always have an associated type stored as additional information. This type represents the subgoal which has to be proved in order to eliminate the hole. Holes will be displayed as follows:

$$\text{case } p = ? \quad \mathcal{P}(p) \equiv \begin{array}{l} \ldots \\ \mathcal{F}(P) \end{array}$$

where P is the subgoal to be proved. Intuitively, the horizontal dots indicate very clearly that the proof is incomplete.

For instance, using typed holes $?_1 : A$ and $?_2 : B$, the partial proof

$$\lambda I{:}A \Rightarrow \neg B.\ I\ ?_1\ ?_2$$

of $A \wedge B$ can be displayed as:

$$\begin{array}{ll} & \ldots \\ (1) & A \\ & \ldots \\ (2) & B \\ (3) & A \wedge B \end{array}$$

if we hide the assumption $A \Rightarrow \neg B$.

14.1.3 Displaying programs

Since our derivation system for programs was based directly on the language *While*, displaying programs is easier than displaying formulas and proofs. However, programs in our formalism contain proof-terms which should not be displayed entirely within the program. If we (1) use line-breaks between composed statements, (2) add indentation to the display of block statements, (3) display expressions in assignments etc. as described above and (4) omit displaying fake-statements we already get a decent display of programs. For instance, a program computing the factorial of natural numbers would be displayed as:

```
|[var n := 0 : nat •
    x := 1;
    while n <> N do
        n := n + 1;
        x := x * n
    od
]|
```

which meets the specification $\neg\perp \triangleright x = N!$ for constant $N : nat$ and program variable x.

Two additions need to be made to this display: (1) we need to display partial programs and (2) we want annotations to be displayed at useful locations. We will discuss these features in the next paragraphs.

Displaying partial programs is done exactly like displaying partial proofs: If ? denotes a hole in a program with sub-specification $P \triangleright Q$, we will display it as

$$\begin{array}{l} \{\mathcal{F}(P)\} \\ \ldots \\ \{\mathcal{F}(Q)\} \end{array}$$

Clearly, the precondition and postcondition of the hole must be displayed, since otherwise the programmer would not be able to complete the program. Ellipsis indicate again the hole.

Displaying annotations at selected locations in the program is established by additional information stored in the nodes of the tree-representation of the program. In this case, annotation is used to denote independently whether the pre- and postcondition of a statement should be displayed.

As an example, consider the unfinished factorial program, without the $n := n + 1$ statement, which can now be displayed as:

```
{$\neg \bot$}
|[**var** $n := 0 : nat$ •
    $x := 1$;
    {$x = n!$}
    **while** $n <> N$ **do**
        {$x = n! \wedge \neg(n = N)$}
        ...
        {$x * n = n!$}
        $x := x * n$
    **od**
    {$x = n! \wedge n = N$}
]|
{$x = N!$}
```

Note that $\neg(n = N)$ is a propositional simplification of the equation $n <> N = true$ which (according to the Hoare rule *While*) is the precondition of the body of the **while**-statement. Apparently, the body of the **while** statement so far is **fake** $p; ?; x := x * n$, where $p : (x = n! \wedge n <> N =$

true) $\Rightarrow$ $(x = n! \wedge \neg(n = N))$ and $? : x = n! \wedge \neg(n = N) \triangleright x * n = n!$. Similarly, there must be a **fake** statement establishing the postcondition $x = n! \wedge n = N$ from $x = n! \wedge n <> N =$ **false**. These kind of simplifications of boolean equations will be generated and inserted automatically by Cocktail as we will see later.

14.2 Interaction

Now that we have specified how proofs and programs are displayed, we need to specify how a user should interact with them. Most theorem provers use a command-line interface as a basis (e.g. [Coq97, Zwa97]), but more recently, graphical user interfaces (GUIs) are optional (see also [TBK92]). Often, these GUIs only support proof-construction in a specific style (e.g. proof-by-pointing [BKT94]) or merely allow the user to invoke the command-line tactics through menus and use non-intuitive structure editors to provide command arguments. Since we want Cocktail to be a user friendly system that invites users to actually use it, we want a GUI that is intuitive and allows the user to construct a proof in her own style. To facilitate this, we will use interaction mechanisms offered by modern GUI-toolkits, i.e. scroll-down menus (context sensitive) pop-up menus, dialogs, drag-and-drop actions etc.

For GUIs, however, there are no formal ways to specify a design. This is mainly caused by the complexity of graphics and interactions. Also, the event-driven character of these interfaces are not formally understood well enough to provide formal support for their design. Therefore, in what follows, we will not attempt to formally specify a GUI or to derive a GUI from the user requirements. Instead, we will describe the motivations that led to the design actually implemented in Cocktail, thereby explaining why certain choices have been made. We hope to have succeeded in presenting our motivations in a sensible way. In practice, design decisions are often withdrawn when it turns out that the result is not satisfying the user's expectations. The design of Cocktail's user interface has not yet needed to undergo such re-design, which gives hope that we have made the right decisions.

Considering how Cocktail displays proofs, the user sees two kinds of information: (1) What has to be proved (i.e. the goals) and (2) what is already known (i.e. axioms, proved theorems and hypotheses). The latter part of information is further divided into two parts: global information available in all proofs to be constructed and local information only available within the current proof. The local information is usually more closely related to the task at hand than the global information. Hence, although global information must also be available, the user will interact more with the local information, which therefore deserves more attention.

The user uses information items to gradually solve the proof obligations. We can distinguish four kinds of possible operations:

- Operations that only require a goal (e.g. decomposing a proof obligation for $A \wedge B$ into proof obligations for A and B).
- Operations that refine a goal using an information item (e.g. refining a proof obligation B to a proof obligation for A, using the information $A \Rightarrow B$ or $A = B$).
- Operations that dissect one information item into smaller items (e.g. concluding A and B separately from the fact $A \wedge B$). This also is a form of forward reasoning, since the goal is not used to draw the conclusions.
- Operations that extract new information from existing information (forward reasoning, e.g. $A \vee B$ and $\neg B$ allow the conclusion of A).

Since the user will spend most of her time working towards a specific goal, we choose to use the concept of a focus, which will always be on one of the (sub)goals. This focus should be selected intuitively by the user. In most applications selecting items is done by pressing the left mouse button, so we will adopt this in Cocktail to select the focus.

In many applications the left mouse button automatically invokes a standard action. For goals this is not desirable, since more than one option can exist. We do not want the user to get irritated by undesired automatic invocation of an action whenever she selects a new focus (in fact, there is enough software famous for suffering from such behavior). A default action can be performed, however, when the user selects an information item in the displayed proof (i.e. a locally available item). The most natural default action is to use the item to prove or refine the selected goal. This is best accomplished by the apply-tactic (see 15.3).

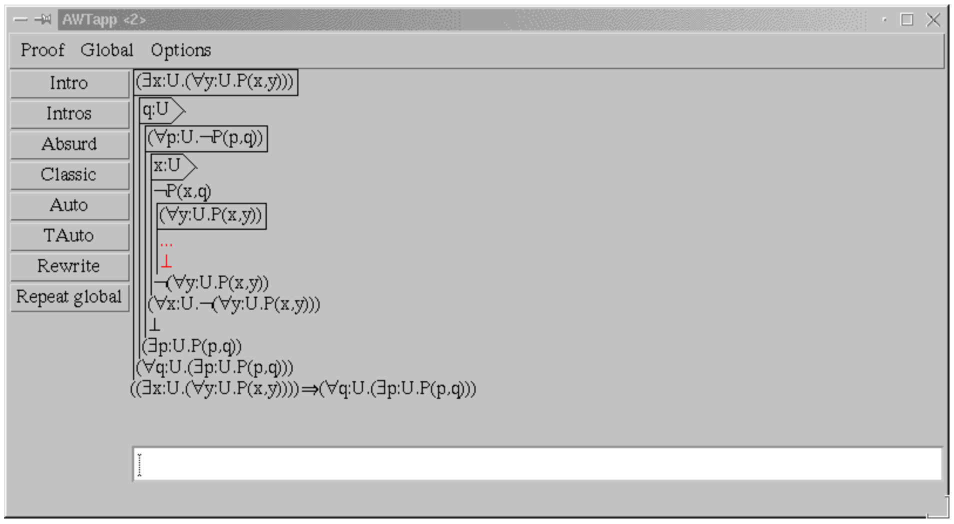
Now that simple mouse clicks have been assigned to simple functionality, we need a useful method to invoke other operations, notably when the user can choose out of several actions. Which actions are available mainly depends on the formula the action is applied to. To avoid presenting the user lots of actions that are not applicable in the case at hand, we use context-sensitive selection menus (Pop-up menus) to let the user select from a list of sensible options. Pop-up menus are triggered in different ways on different platforms. This is supported by Java and used in Cocktail. In all cases, the options shown in the menu depend on the formula under the mouse-cursor at the time of invocation and on the kind of the formula (information or goal). If necessary, the selection of the actions is followed by dialogs to obtain additional information (e.g. what rewrite-rule to use for Leibniz equality etc).

Some rules (like **classic**) can always be applied to any goal. Since menus become confusing when they contain too many items, these rules are invoked by buttons that are displayed to the left of the proof. This is only possible for goal-based actions, since information items cannot be selected and hence, the action associated with pressing a button cannot determine on which item it should be applied.

By now, using menus, all actions can be invoked, but there is still room for improvement. Simply clicking on an information item invokes the most natural action for the information, given a selected goal. Equally, there is a most natural action to combine two information items, namely to derive new information from the two items given. For instance, given information items stating $A \vee B$ and $\neg B$ combining them would naturally lead to an information item stating A. Combining a pair of information items is easily done by drag-and-drop: the first item is dragged to the second item.

All actions so far only apply to locally available information items. Since globally available items are displayed only once in a separate window, we cannot directly link actions on them to goal directed tactics, since it is not clear which goal is targeted when more proofs are constructed simultaneously. Instead, we choose to have one selected global item, which can be imported into local proofs (i.e. repeated). Once the item is visible in the local proof, all actions mentioned above can be used on it. Repetition can be used at any moment and hence, is invoked by a button to the left of the display.

This GUI has been implemented in Cocktail for the theorem proving part and turns out to work pleasantly. Figure 14.3 shows a screen-shot of a proof under construction.



```
(∃x:U.(∀y:U.P(x,y)))
 q:U
  (∀p:U.¬P(p,q))
   x:U
   ¬P(x,q)
    (∀y:U.P(x,y))
    ...
    ⊥
   ¬(∀y:U.P(x,y))
  (∀x:U.¬(∀y:U.P(x,y)))
  ⊥
 (∃p:U.P(p,q))
(∀q:U.(∃p:U.P(p,q)))
((∃x:U.(∀y:U.P(x,y))))⇒(∀q:U.(∃p:U.P(p,q)))
```

Figure 14.3: A proof under construction in Cocktail. On screen, the current focus is displayed in red.

Chapter 15

Implementation

Cocktail has been implemented in Java. This chapter describes the most important classes and their categorization in packages. We assume the reader is already familiar with imperative object oriented programming languages. Nevertheless, we will discuss a few concepts specific to Java in the following paragraph.

Java is an object oriented language based on C++. To avoid many of the problems inherent of C++, many improvements and simplifications have been made in the design of Java. For instance, Java does not allow pointer-calculations and handles dynamic memory allocation and de-allocation automatically. Objects that are no longer used are detected by a garbage collector and removed automatically, thereby eliminating the possibilities for dangling pointers and memory leaks. Also, Java has many standard libraries, available for all platforms, to enable networking, graphical user-

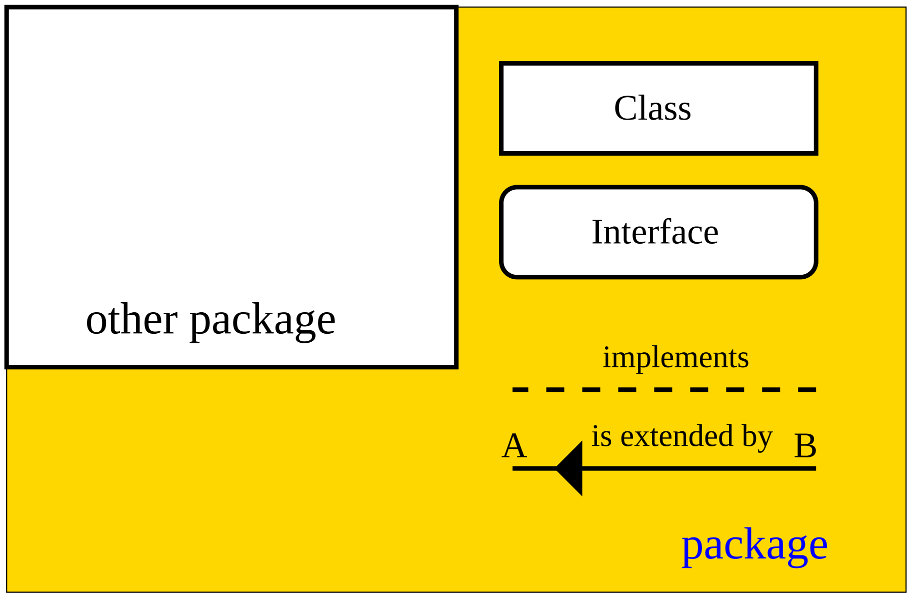


Figure 15.1: The symbols used for schematics of the class structures.

interfaces and file-access.

Multiple inheritance is not supported by Java. Instead, the concept of Interfaces is introduced. An Interface in Java is just a class description in which none of the specified methods is really implemented. Also, data-fields are not allowed, except for constants. Interfaces can inherit from an arbitrary number of other interfaces. A class can inherit from no more than one other class, but it may implement many interfaces. The interfaces implemented must be declared explicitly and the class must overrule all methods declared in its interfaces.

In Cocktail, most interfaces are used to define constants representing types of messages. These interfaces will not be discussed in detail. We will only discuss interfaces used for other purposes.

Along with the descriptions, we also depict Cocktail's class hierarchies schematically, using a diagram notation similar to the one used in [Fla97]. A legend is given in figure 15.1.

15.1 Infrastructure

The infrastructure used by Cocktail was designed by C. Hemerik. Conceptually, the infrastructure deals with messages sent between a number of tasks. These messages are sent over a bus, of which the user of the infrastructure needs not to be aware. The bus is managed by a special task called the *director*. The director is the only entity that is allowed to create new tasks and register them on the bus or that can remove tasks from the bus. Tasks are identified by a unique number that is assigned upon creation. Instead of explicitly sending a message over the bus, each task has methods to send a message to (1) a specific recipient, (2) all recipients interested in messages from this task, (3) all existing tasks, (4) the director.

For example, consider a text editor to be used to write large programs. Such a text editor must be able to deal with several files at once, some of which may be visible in several views. Using the infrastructure mentioned above, this can relatively easy be achieved by creating one task for each file being edited and one task for each view. Every command given by the user of the editor is initially received by a view (since the view is the visible part on the screen) and then sent directly to the task holding the corresponding file, using a method of type (1). Hence, even if the user alternately uses different views to edit a single file, there is one task ensuring that all changes are made to a single file. However, there are also messages sent from the non-visible tasks to the views: every time the file changes, all views need to be updated. Therefore, the non-visible task holding the file sends a message to all its views, using a method of type (2). If the user sends a command to save all files, a message is sent using a method of type (3). In order to close a view, the view itself sends a message to the director with a method of type (4). The director then removes the task and if it was the last view showing a certain file, it will also close the task holding the file.

Cocktail uses the infrastructure in a similar way, except that it does not deal with several files being edited, but with several proofs and program derivations in progress.

How the infrastructure is implemented in Java is depicted in figure 15.2. We describe the added functionality for each class and interface briefly:

BusMessage This simple class represents the messages that are sent over the bus. It contains the identity of the sender, the recipient (if one is specified), the type of message (one-to-one, one-to-subscribers, one-

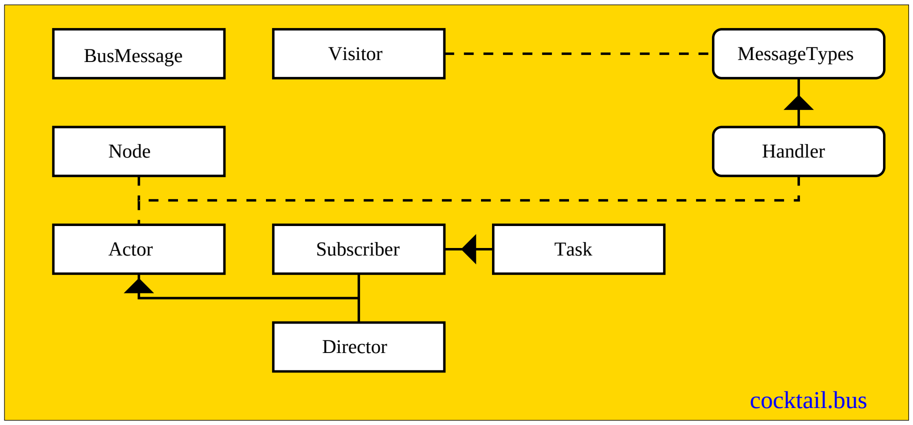


Figure 15.2: The class structure of the bus-system used by Cocktail.

to-all or one-to-director), a message-code and a field for additional message-information (e.g. command arguments).

MessageTypes The constants to distinguish between messages sent to a single recipient, all tasks, subscribers etc. are defined in this interface.

Handler This interface specifies methods that must be implemented by all classes that want to deal with messages. It provides methods to check if the message should be passed to the object, to check if the message is valid for the object and to send the message to the object. Since all its implementations will be able to send messages over the bus, this interface extends the MessageTypes interface.

Actor The actor class is the simplest class that can send messages over the bus. It implements the handler interface. It also provides the methods to send messages over the bus, such that the bus no longer has to be addressed directly by its descendents.

Subscriber This is a direct descendent of Actor. Subscribers are objects that can subscribe to other tasks on the bus (i.e. they will be interested in all messages these tasks send to their subscribers). Hence, a subscriber maintains a list of task identities in whose messages it will be interested.

Director This special inhabitant of the bus is the entity that controls all access to the bus. It should be created during initialization of the system. After creating the director, all tasks on the bus should be created by calls to the director. Descendents of Director will be needed to add the capability to create specific tasks implemented by descendents of Task. Of all existing tasks, the director will focus on one. What it means for a task to have the focus, depends on the use of the infrastructure in Director's descendents. Since the director is not subscribing itself to other tasks, it is a direct descendent of Actor.

Task This descendent of Subscriber is the parent of all tasks that really do the work. It also stores a name for the task, which can be used to identify the task to the user (e.g. a filename). After all, a user wants more specific information about a task than an ID-number.

Node Many applications will use tree-structures. The Node class provides methods to utilize most operations usually performed on trees. To actually build any meaningful trees, Node must be extended by classes implementing methods like `NrOfSons` returning the number of sons of the node, `Son(i:integer)` returning son number `i` etc. Most importantly, the node class supports the Visitor pattern by means of messages traveling through the tree (which are represented again by the BusMessage class). Therefore, the Node class implements the Handler interface. Often, a message traveling through a tree contains a Visitor object as extra information. This Visitor is passed to the visited node and can perform some action on this node (e.g. gathering some information about the node, changing some values stored in node labels etc.)

Furthermore, a Node can be annotated with extra information. The methods to add, extract and check for the availability of information are provided, but the actual storage of this information must be implemented by its descendents. This ensures a uniform application programming interface for information stored in nodes.

Visitor This class represents the top-class of all visitors. These visitors are sent through a tree using a BusMessage and the appropriate methods of Node. In order to determine the action performed by a specific visitor, one has to extend the Visitor class and re-implement the Visit method that accepts the visited Node as an argument. Also, one has to re-implement the Condition method, accepting a Node argument and returning a boolean whether or not the visitor needs to perform an action on the visited node.

15.2 Symbolic Engine

The symbolic engine consists of (1) a representation of type judgments $\Gamma \vdash A{:}B$ for $\lambda P-$ and programs and (2) an algorithm to check the correctness of these judgments. Once we have these, the tool has to let the user create type judgments by using tactics and use the type checker to verify their correctness. Using this approach, correctness of the final result is fully ensured by the type checker, regardless of the complexity of the tactics used to construct the type judgments.

In Cocktail, all terms are represented by trees. For this, a number of descendents of the Node class are defined: Name, representing the name of a variable or constant; Term, being the superclass of all terms used and Item, which is a representation of Name$\times$Term to denote typed variables. An independent class ItemCollection is used to represent lists of Items, like contexts. The Term class overrules the SetAnno and GetAnno methods inherited from Node to allow annotation of terms with their type, display and hiding information.

In turn, Term has three more descendents, used to add functionality used by several subclasses: BindingTerm, which introduces a bound, typed variable that can be used within its body (i.e. BindingTerm represents Item$\times$Term); RefTerm, which represents a reference to a global or bound variable by storing a reference to the corresponding binding Item; and ChainTerm, which represents a list of terms (e.g. the argument list of an instantiated parametric constant). All of these classes implement the clone method, returning an exact copy of the tree they represent. However, copying terms with bound variables deserves special attention. If RefTerm objects occurring in the body of a BindingTerm would be copied exactly, then the copy of the bindingterm would contain in its body references to the variables bound by the original bindingterm (see figure 15.3).

In order to create correct copies of bindingterms, the Item class was extended to contain an auxiliary reference, which refers to a copy of the item during copying of bindingterms and to null otherwise. When a refterm is requested to provide a copy of itself, it will check if the auxiliary field of its corresponding item refers to a copy of the item. If so, a refterm based on the item's copy is returned. If not, an exact copy of the refterm is returned. Copying a bindingterm is correctly done in the following way (see figure 15.4):

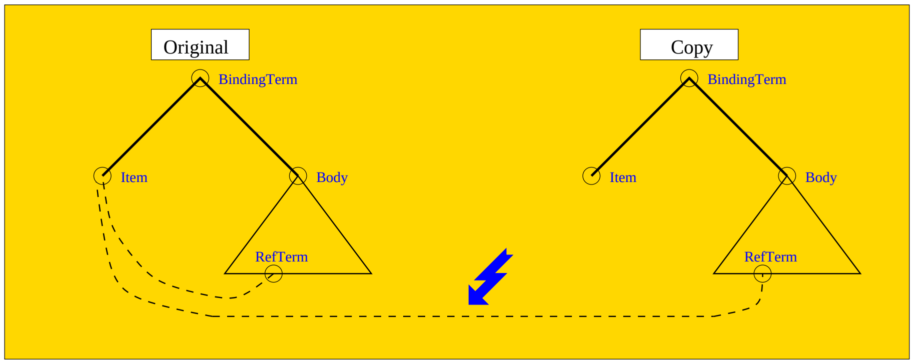


Figure 15.3: When creating copies of bindingterms, using exact copies of item and body yields incorrect references to bound variables.

1. Create a copy of the BindingTerm's item and set the auxiliary field of the original item to this copy.

2. Create a copy of the BindingTerm's body. Due to the modified way in which RefTerms are copied, all references to the bindingterm's item are replaced by references to the copied item in the body's copy.

3. Create the new bindingterm, using the item's copy and the body's copy. The auxiliary field in the original item is restored to null.

Terms also provide a variant of the clone method, called Substitute. Substitute requires two arguments: an Item and a Term. It returns a copy of the original term, in which all references to the given item are replaced by copies of the given term. The implementation of Substitute is similar to the implementation of clone, except that Substitutes of child nodes of the terms are used instead of copies.

Based on these classes, derived classes are used to represent directly the syntactic structure of $\lambda P-$ and *While*; e.g. we have a class Term_Pi derived from BindingTerm representing Π-terms etc. For a complete overview, see figures 15.5 and 15.6.

Now that we can represent terms and contexts of $\lambda P-$ and *While*, we can build a type checker. Instead of type-checking, Cocktail constructs the type of a given term from scratch, causing an error if the term is not typeable. If a type for a term is constructed and the term was not already typed, the

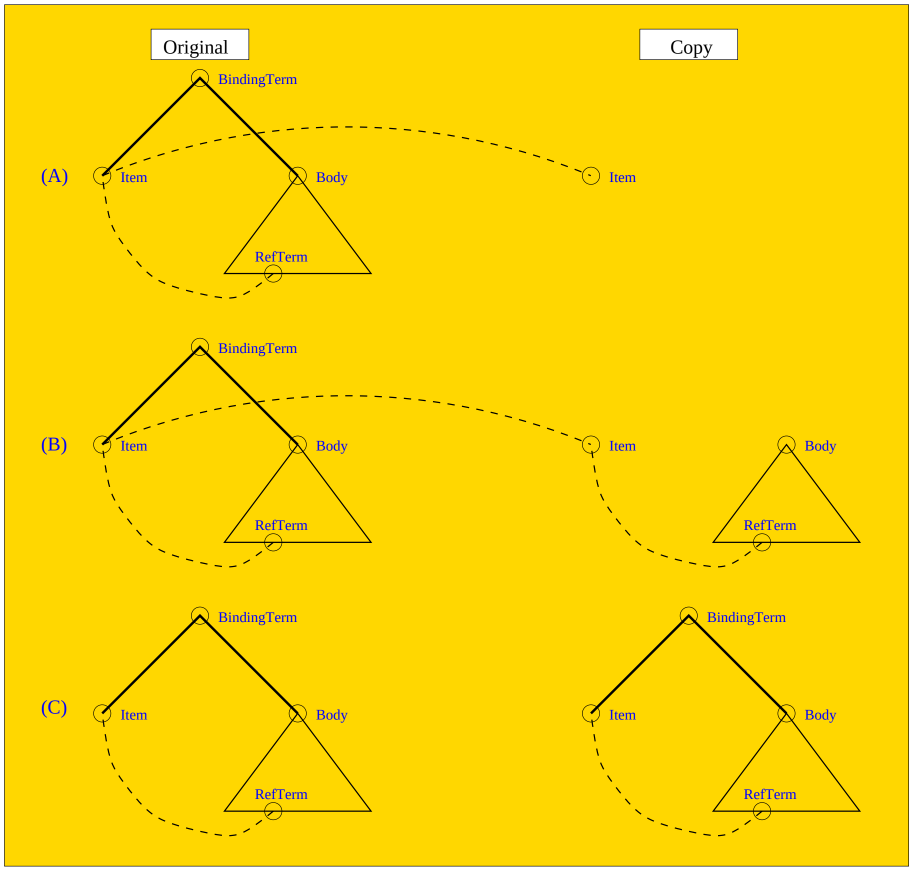


Figure 15.4: Copying bindingterms correctly in three steps: (a) copy item and create an auxiliary reference, (b) copy body, taking care of refterms, (c) create the bindingterm's copy and delete the auxiliary reference.

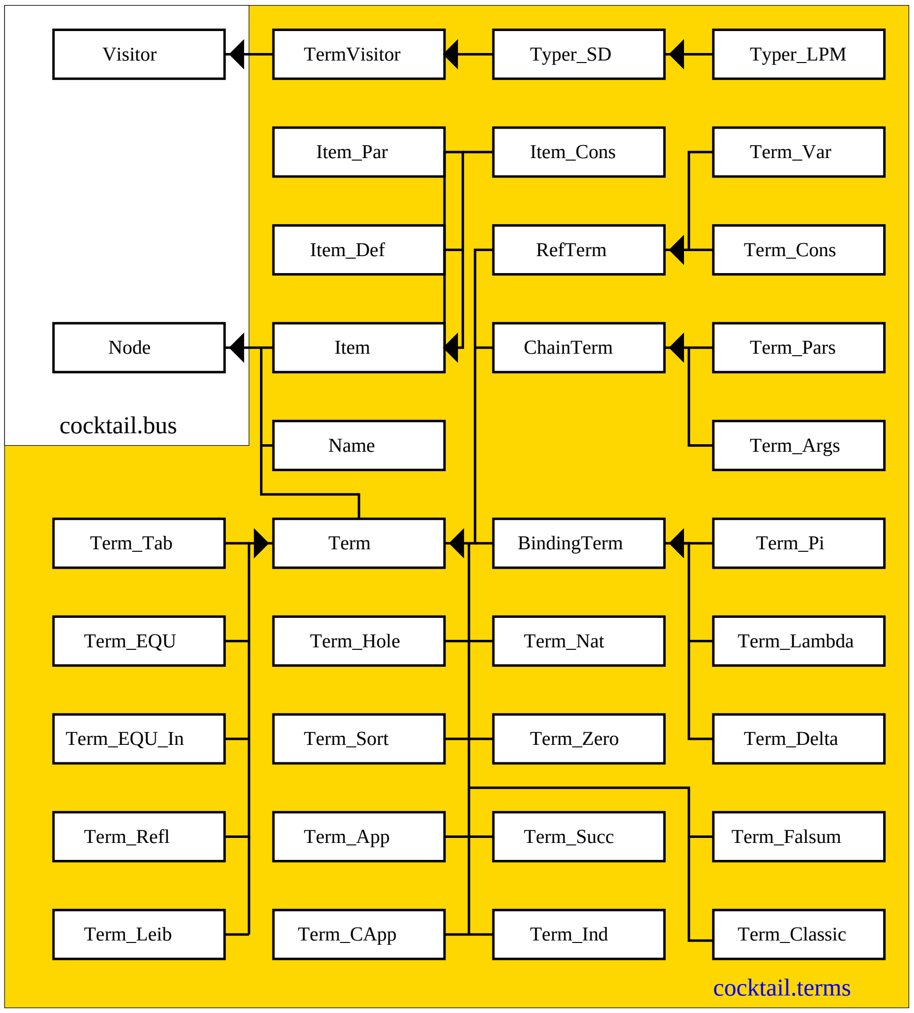


Figure 15.5: The class structure to represent formulas and proofs in Cocktail and to compute their types

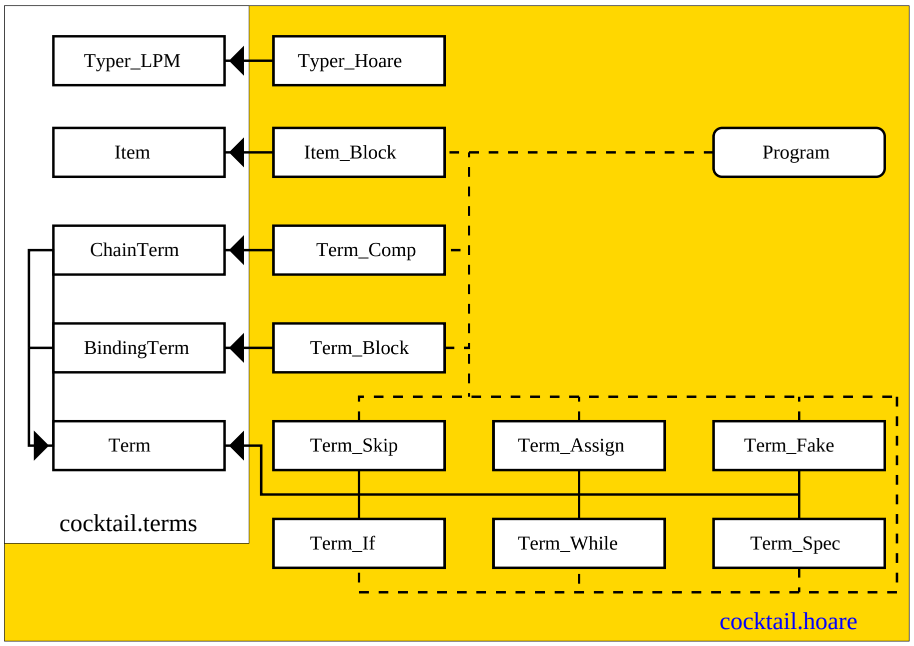


Figure 15.6: The class structure to represent programs in Cocktail and to compute their specification. All terms in this package implement the empty Hoare interface to distinct them from formulas and proofs.

type is added as annotation to the term. If a type was already given for the term, the constructed type is compared to the given type, leaving the original type intact in order not to destroy any annotation stored in the term's type.

The type of a term is constructed bottom-up: First the types of all subterms of the given term are computed and stored as annotation in their corresponding nodes. Then the type of the given term is computed from these types, checking all preliminaries of the PTS-rule used to obtain the original term. Since the conversion rule is not present in $\lambda P-$, there is never more than one rule to consider (type checking is entirely syntax-directed). Cocktail does support the definition mechanism for PTSs described in section 6.3 and hence, special care has to be taken for definition terms. This is eased by the restriction that Cocktail only uses definitions in proof terms and the global context. That is, a term $x = a : A$ in b can only occur for $A : *_p$ and $b : B$ with $B : *_p$ in the corresponding contexts. Since a proposition P can never depend on a proof term, this restricted use implies that $x = a : A$ in $P =_\delta P$. Therefore, the type constructed for any definition term $x = a : A$ in b will be B, rather than $x = a : A$ in B. The definition terms in the context are used to store proved theorems. For example, if in global context Γ the user proved $\Gamma \vdash p{:}P$ with $P : *_p$, then the global context is extended with $H = p : P$, where H is the name of the proved theorem.

Although it seems advantageous to allow definitions in large formulas as well, we do not use them in that way. If we would, unification and type checking would become less efficient, since we need a δ-expansion rule in the logic. The tool's flexible hiding mechanism for terms can easily be extended to allow shorthand notations to be used for formulas, without the need for PTS definitions.

The type constructor is implemented in classes TermVisitor, Typer_SD and Typer_LPM. The termvisitor class implements the Handle method such that for every kind of visited node used in a regular CPTS, a suitable method is called. Also, it provides dummies (empty method implementations) for all of these methods. This class will also be used for other purposes than type construction later on. The Typer_SD class (SD stands for Syntax-Directed) inherits from TermVisitor and implements the methods for each type of node such that their types are computed and stored. To check the correctness of Π-types, constants and sorts, Typer_SD uses a general representation of a CPTS, which can easily be replaced by another CPTS. The definition of $\lambda P-$ is loaded by default. The classes for this CPTS representation will not be discussed any further. Finally, Typer_LPM (LPM stands for Lambda P Minus) extends Typer_SD to deal with terms that only exist in $\lambda P-$ (e.g. natural induction, Leibniz equality etc.)

To check the correctness of programs, Typer_LPM is extended by the class Typer_Hoare. Note that all methods of Typer_LPM are needed to check the correctness of fake-statements.

15.3 Tactics

Now that we can represent type judgments and check their correctness, we must provide editors to create these terms, i.e. we need formula editors to write theorem and program specifications, proof editors to prove logical theorems and program editors to derive programs. All smart ways of proof construction will be implemented in this part, including the tableau-based automated theorem prover. However, due to the type-checker, correctness of the final result will always be ensured (or at least errors are detected in time).

Terms and contexts are maintained by subclasses of Editor, which in turn is a subclass of Task. Editors perform all the operations allowed on the information they maintain. In general, they notify their subscribers of all changes made. For instance, if a proof is further refined, all subscribers are notified so they can react to the changes (e.g. update the display). Note that editors do not provide a user-interface. They provide the means to construct and edit terms through sending messages.

The class Manager is a marker class, used as the superclass of all editors that maintain a global source of the system. It has subclasses Manager_Context and Manager_PTS, that maintain the global context and the specified PTS respectively. All changes made to the global context, are caused by messages sent to Manager_Context. Like the Director, only one instance of each type of Manager should be present on the bus. However, Managers are not allowed to create new tasks registered to the bus.

Editor_Node is a descendent of Editor which maintains information about a single tree. Within this tree one node is focussed to determine where all actions are targeted. Editor_Node maintains the following information about the tree:

- The frontier: a list of all leaves of the tree in the order they appear. If one of the leaves is focussed, its index in the frontier is also computed.
- A hole list, which is a sublist of the frontier and lists all leaves that are holes (holes are leaves by definition). Also, if the focus is a hole, the index in the hole list is computed.
- The path, being the list starting at the root node, ending with the focussed node, in which every node is a child of the previous node.

Editor_Term is an extension of Editor_Node, which also maintains a local context consisting of all items of BindingTerms found on the focus's path.

Editor_Lambda, a descendent of Editor_Term, accepts messages to replace the focussed node F by propositional constructs, such as $F \wedge ?_1$, where $?_1$ is a newly created hole.

Editor_Proof, extending Editor_Lambda, is the first editor in the hierarchy that takes the type of the focussed node into account. It provides tactics to construct proofs of theorems that are stored as the type of the root node. Initially, the root node will be a single hole. This hole will be replaced by λ-terms that possibly contain new holes, each annotated with a type representing a new subgoal. A theorem is proved if the tree contains no more holes.

To replace a hole by a λ-term, the Editor_Proof constructs terms based on the type of the hole. For instance, the intro-tactic replaces a hole typed as $\Pi A{:}B.\ C$ by a λ-term $\lambda A{:}B.\ ?_n$, where $?_n$ is a new hole annotated with type C. Note that the type of this λ-term is equal to the type of the original hole, provided that $?_n$ is replaced by a λ-term of the type indicated by its annotation.

Where necessary, the λ-terms constructed are annotated with display information to hide parts of the proof as explained in section 14.1.2. For instance, a hole annotated with $A \wedge B$ may be replaced by the λ-term

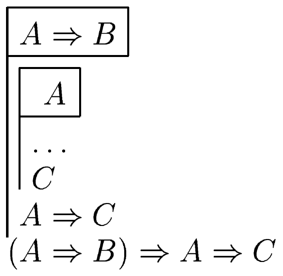
$$(\lambda H{:}A \Rightarrow \neg B.\ H\ ?_a\ ?_b),$$

where $?_a$ and $?_b$ are annotated with types A and B respectively. The assumption $A \Rightarrow \neg B$, however, is annotated to be hidden on screen (see the example on page 171 of section 14.1.2).

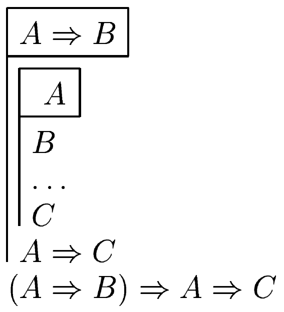
Forward reasoning is implemented by inserting local definitions in the proof term. For instance, suppose we have the partial proof

$$\lambda H1{:}A \Rightarrow B.\ \lambda H2{:}A.\ ?,$$

where ? is annotated with type C, the user will see the following:



When (s)he drags the hypothesis A to the hypothesis $A \Rightarrow B$, Cocktail will combine $H1$ and $H2$ by an application $H1\ H2$ of type B. This term is inserted as a definition $H3 = H1\ H2 : B$ in ? at the place of the hole to yield $\lambda H1{:}A \Rightarrow B.\ \lambda H2{:}A.\ H3 = H1\ H2 : B$ in ?, displayed as:



Although most tactics are straightforward to implement, some require more complex computations, usually using unification. Of the latter ones, the most important tactic, Apply, is described here in more detail.

The apply tactic constructs a proof term for goal P, using an information item $A : B$ passed as the tactic's argument. In order for apply to work, P must be an instantiation of B. That is, if B is of the form $\Pi v_1 : B_1 \dots \Pi v_n : B_n.B'$, then P must be equal to $B'[v_i := e_i]_{i=1}^{n}$ for certain expressions e_i, $i \in \{1, \dots, n\}$. The λ-term $A\ e_1 \dots e_n$ will then have type P as requested. These expressions e_i can be found using unification on B' and P. However, even if B' and P unify, the apply tactic may fail if no substitute is found for one or more v_i with $B_i : *_s$. For instance, applying $\forall p, q, r : nat.p < q \wedge q < r \Rightarrow p < r$ to $a < c$ yields substitution $[p \mapsto a, r \mapsto c]$, but fails since no value for q is given. On the other hand, there will never be values for v_i with $B_i : *_p$, since these can never occur in B'. These B_i will become the new subgoals. Hence, for each v_i with $B_i : *_p$, we create a hole $?_i$ annotated with type $B_i[v_j := e_j]_{j=1}^{i-1}$ and use it instead of e_i.

For instance, if the current goals is to prove $Q(y)$ for some $y : U$, and the user applies an information item stating $H : \forall x : U.P(x) \Rightarrow Q(x)$, then $Q(x)$ is unified with $Q(y)$, yielding unifier $[x \mapsto y]$. A hole, annotated with type $P(y)$ is created, say $?_1$. Now the term $H\ y\ ?_1$ has type $Q(y)$ and is substituted for the original hole. The new goal is now $P(y)$, as was expected.

Similar approaches are used to compute proof terms for apply-forward, Leibniz equality etc. It is always checked if universally quantified variables can be instantiated through unification. If so, hypotheses are replaced by fresh holes. For instance, rewriting a goal $C(y)$ to $B(y)$, using hypothesis $H : \forall x : nat.A(x) \Rightarrow (B(x) = C(x))$ will succeed (for the symbol '=', see section 12.2.1), resulting in subgoals $B(y)$ and $A(y)$. Compared to most other systems, this is a very general rewrite tactic.

In practice, rewriting is used very much and often for the same purpose, like rewriting a while statement's postcondition $I \wedge g = \textbf{false}$ into $I \wedge G$, where G is a propositional version of the guard g. To ease this type of rewriting, a user can compose lists of rewrite rules. Such a list contains a number of (quantified) equalities taken from the hypotheses and proofs of the global context in a certain order. Also, a direction is specified for each rule in the system. Cocktail can apply such a set of rules instead of a single rule, by repetitively applying the rules in the system in the given order and direction until no more rules apply. Since no checks are performed to test the termination properties of the specified rewrite systems, the user must be careful when constructing them. For instance, one might use a rewrite system to compute plus, times and factorial expressions. Since rewrite-systems cannot be considered separate from the global context of which they obtain their individual rewrite rules, the set of rewrite systems is stored in the Manager_Context, which also maintains it.

Another elaborate tactic is the Auto tactic, which implements the tableau-based automated theorem prover. This tactic is implemented in the class Editor_Tableau in the auto-package (see figure 15.7). This editor extends Editor_Node and builds a tableau as a tree consisting of Node-descendents Alpha_Node till Special_Node as soon as a theorem is provided. Instances of TabNode are used to store labels of leaves which yet have to be expanded. In order to force termination, the user may limit the maximum amount of leaves, the maximum depth of the tableau and even the maximum amount of time spent in constructing a tableau. When finished, the Editor_Tableau object contains the tableau and, if it is closed, can compute the corresponding lambda-term on request. The TAuto tactic uses the same theorem prover, but initialises it with a start label that only contains the local context, which often yields more efficient proofs.

Editor_Hoare is a descendent of Editor_Lambda, which implements tactics for deriving programs. Although Editor_Hoare's tactics are rather different from those of Editor_Lambda, no complex programming was required to implement this class. For example, the tactic that replaces a constant N by a variable n in postcondition Q takes two arguments: a template T, being the postcondition Q in which several occurrences of a constant are replaced by a special token $\odot$ (in the implementation the only hole with number 0) and an assignment $n := e$ specifying the name and initial value of the variable. The

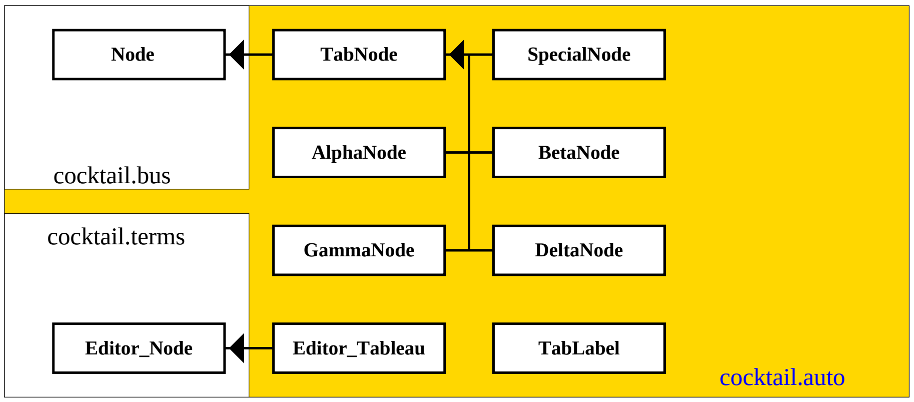


Figure 15.7: Classes needed to construct tableaux.

tactic first constructs a λ-term p of type $(T[\odot := n] \wedge n = N) \Rightarrow Q$, which is easy since $Q \equiv T[\odot := N]$ and we have Leibniz equality (see section 12.2.1). Then the program term $|[\textbf{var}\ n := e : U \bullet ?_1; \textbf{fake}\ p]|$ is constructed where $?_1$ is specified as $P \wedge n = e \triangleright T[\odot := n] \wedge n = N$, where P is the precondition of the original hole. Obviously, the program has indeed the specification $P \triangleright Q$.

If the user specifies rewrite systems named 'bool2prop' and 'prop2bool', Editor_Hoare will use these automatically. Bool2prop will be used to automatically rewrite a guard g of an if- or while-statement into a more suitable proposition than $g = \textbf{true}$ or $g = \textbf{false}$. Prop2bool is used during the construction of while-statements and should attempt to rewrite a proposition to the form $g = \textbf{false}$, where g will become the guard of the while statement.

Similarly, Editor_Hoare deals with other tactics, creating proof terms for fake statements to use the Hoare logic more flexibly. In particular, the tactic close invokes an Editor_Proof to let the user interactively prove $P \Rightarrow Q$ to generate a fake statement satisfying $P \triangleright Q$ that exactly closes a hole. A program is finished when no more holes exist.

An overview of the editor-classes is given in figure 15.8

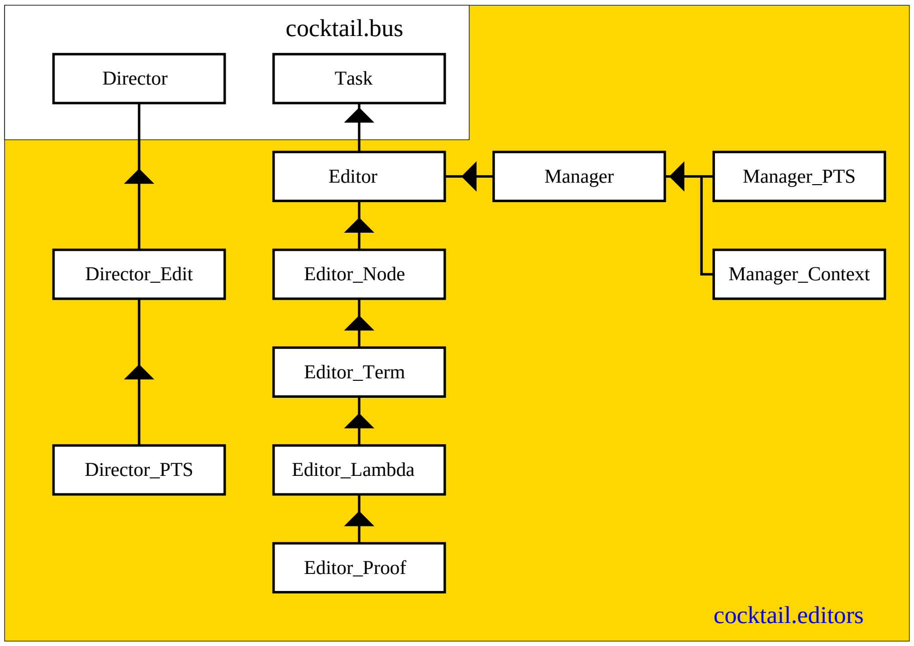


Figure 15.8: Editors and Directors in the editors-package.

15.4 Formula and Program Display

This section describes how formulas, represented in Cocktail by trees, are displayed on the screen. We also take into account that the graphical user interface lets the user interact with displayed proofs and provide the information needed to do so during displaying.

Cocktail's display method makes a distinction between a display and a displayer. A display is used to make information visible to the user, whereas a displayer is used to convert internal representations of data into displayable format, which is then sent to the display.

In the implementation, each display must implement the TermDisplay interface, which specifies methods to append a string to the display, start a new line and increase or decrease the indentation. A display is passed to the TermDisplayer-object during initialization. The TermDisplayer is an extension of TermVisitor, that calls the Display's methods while traversing a formula's tree. Whenever a string is appended to the display, the node to which this display-string refers is passed to the display too. The display can use this information during interaction, when it is necessary to link a screen location to a sub-term in the tree (see also section 15.6). The TermDisplayer uses the notations specified by the user to display parametric constants and the display-annotation in the nodes to display propositional constructs.

A notable class implementing the TermDisplay interface is StringDisplay, which does not display the formula at all. Instead, all strings sent to StringDisplay are concatenated to form a single formatted string for the formula. This string representation can be used to display the formula as part of a bigger whole, like a proof.

TermCanvas is an implementation of TermDisplay that displays terms directly onto the screen. It does not store the visual layout, hence to redraw a term after it has been changed, the TermDisplayer, sending commands to the TermCanvas, has to re-traverse the entire tree. Although this is inefficient, it does not result in an unworkable slow display, since TermCanvas is only used for displaying logical formulas and programs and these tend to remain small in practice.

Programs are displayed in exactly the same way, but use an extension of TermDisplayer called HoareDisplayer. The main difference is, that HoareDisplayer, depending on hiding information in program nodes, can display pre- and postconditions of programs. Note that this information is not part of the program tree, but of its type-annotation.

15.5 Proof Display

The displaying of proofs is done slightly different from displaying formulas and programs. Again, there is a ProofDisplay interface and a ProofDisplayer class extending TermVisitor, distinguishing display from displayer. The ProofDisplay interface does not accept pieces of text, but rather formulas and information items. These are passed through various methods, depending on whether they represent definitions, assumptions, variable-introductions, holes or regular pieces of proof. The ProofDisplayer chooses how to display pieces of proof by the term-structure and the display annotation in the proof's nodes.

Since, in contrast to formulas and programs, proof-terms can become quite large, it is necessary to build a display representation of the proof. Re-traversing the proof term during scrolling would slow Cocktail down unacceptably.

In ProofCanvas, the only implementation of ProofDisplay so far, each item to be displayed is stored in a ProofLine object. Each ProofLine object represents a single line of the proof visible on the screen. It stores the Node to which the displayed line refers, the type of the line (i.e. how it should be displayed) and the depth of the local context (the number of vertical lines drawn before the formula). Once the entire proof has been converted into ProofLines, the proof can be displayed without referring to the proof term. To convert formulas to be displayed in a single proofline, the StringDisplay discussed in the previous section is used, yielding a formatted string. For instance, the example proof of section 14.1.2 on page 170 is converted into the following set of prooflines:

$$\begin{array}{l} \boxed{A} \\ \vert\,\boxed{B} \\ \vert\,\vert\, A \wedge B \\ \vert\, B \Rightarrow A \wedge B \\ A \Rightarrow B \Rightarrow A \wedge B \end{array}$$

(The openings between prooflines are not shown on screen).

15.6 Interaction

The implementation of the user interface described in section 14.2 is mostly a matter of combining widgets and user-interface components from the abstract windowing toolkit (AWT) of java. We will not discuss this in more detail in this thesis.

However, to invoke the actions and menus, Cocktail must determine which sub-terms were selected by the user. Hence, a location on the screen must be mapped onto a sub-term in the representation.

For terms and programs, this is done by repainting the formula, having a termdisplayer traverse the formula's tree. The TermCanvas implementation of the TermDisplay interface keeps track of the location at which data is currently drawn on the screen. At the point when the selected screen location is overwritten by strings belonging to a sub-term, the TermCanvas remembers the node passed along with the string. This node is the root of the selected sub-term. Although this method tends to be slow because of its redrawing, in practice the formulas and programs are small enough to allow real-time interaction.

For proofs, a complete traversal of the term becomes too time consuming. Therefore, the node corresponding to a proof's line on the screen is stored within the ProofLine object. Cocktail can simply compute which line was selected and extract the sub-term from its representation.

The user-interface is implemented by descendents of View. Each View is a task registered on the bus and created by the bus's director that shows a single window on the screen. Basically, the View class provides the basic lay-out of Cocktail's windows: a window with possibly a menu-bar at the top, a panel of buttons to the left, a message-bar at the bottom and a main panel in the middle. The actual menu-bar, button-panel and main-panel are provided by the descendents and can range from a single TermDisplay as the main panel (View_Term) to a split-window to separately display the pre- and postcondition of a program's specification. The full-class overview is depicted in figure 15.9. In general, each View belongs to an editor, whose contents it displays. Conversely, however, one Editor can have several views even though this is not utilized in the current implementation of Cocktail.

The classes View_Term and View_Proof provide the graphical components to show a term or a proof in a window on the screen. The extended classes View_Term_Edit and View_Proof_Editor also provide the user buttons and menus to interact with the terms and proofs. The descendents View_TEdit_Goal and View_TEdit_Spec of View_Term_Edit provide slightly different views for accurate viewing of goals and specifications respectively.

View_Context implements a frame showing the global information of the system (i.e. the global context). View_Context_Edit is the class that ties the entire user-interface together. When the user issues a command to extend the context with a theorem, View_Context_Edit asks the Director of the bus to create an Editor_Term and a View_TEdit_Goal linked to this editor to let the user construct the theorem. In turn, the View_TEdit_Goal will ask the Director to create an Editor_Proof and a corresponding View_Proof_Edit to let the user prove the theorem she just specified. Finally, the View_Proof_Edit sends the proved theorem back to the Manager_Context, which adds it

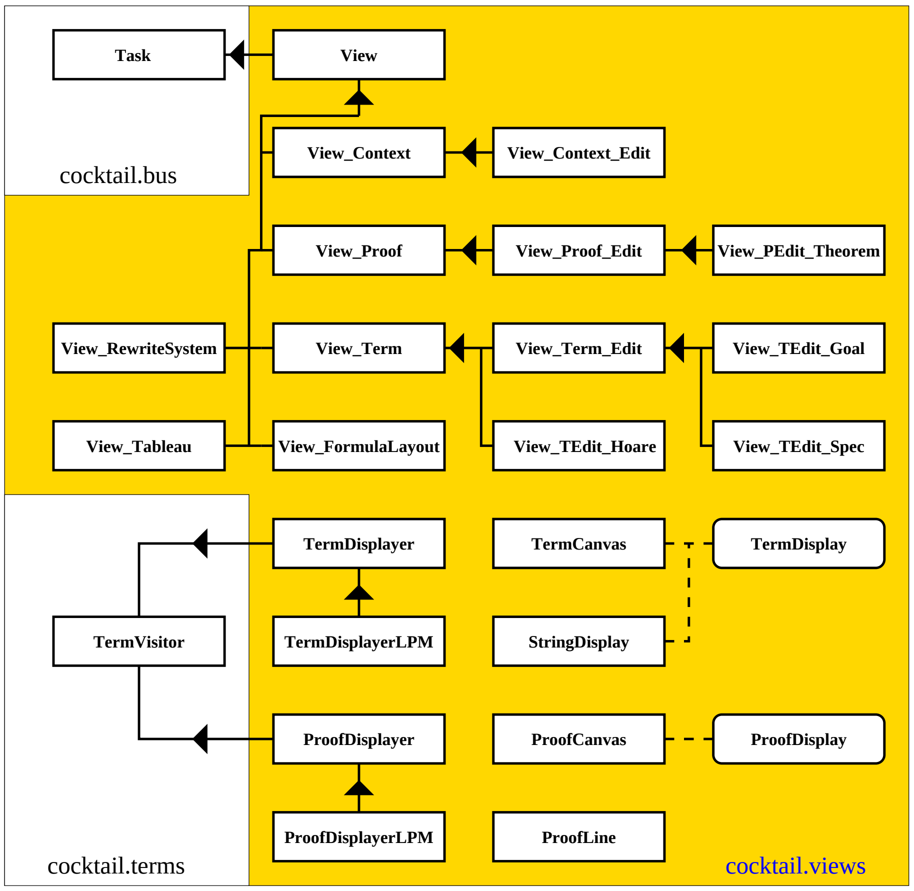


Figure 15.9: Views and GUI-components in the views-package.

to the global context. Since the Manager_Context notifies its subscribers (hence, the View_Context_Edit object we started with), the new theorem automatically shows up on the screen. The classes View_FormulaLayout and View_RewriteSystem are used to view and edit the layouts of formulas and the rewritesystems stored in the Manager_Context respectively.

Chapter 16

Results

This chapter discusses the results of our labour. We look back to see if Cocktail has become what we wanted: A tool for deriving correct programs. Also, we will analyze a few weaknesses still present in the tool and discuss what steps can to be taken to eliminate them.

16.1 Comparison with the Initial Requirements

In order to find out if Cocktail has become what we aimed for, we compare the capabilities of the current version of the tool with the requirements initially put forth by the programmer, the logician and the system's designer.

16.1.1 The Programmer's Requirements

The programmer stated two main requirements in chapter 2:

- The programmer wanted a tool that was mainly focussed on the program. In Cocktail, the focus is indeed on the program. Encoding of the program in a theorem prover's logic is not necessary. The user interface for programs is not based on theorem proving, but on program derivation.

- Also, the programmer wanted proofs to be constructed automatically whenever possible. Theorem proving in the tool is (partially) automated. However, Cocktail's ATP does not yet support equational reasoning.

  In cases where a proof cannot be constructed automatically, the programmer has to construct it herself, but she wants this to be as easy as possible. Interactive theorem proving in our tool is based on first-order logic and includes extensive support for rewriting. The user interface for proof construction is based on natural deduction.

16.1.2 The Logician's Requirements

The logician was mainly concerned about the safety of the tool. She came up with three requirements, which are briefly discussed below.

- The logician wanted the implementation of the systems to be safe and to conform to the De Bruijn criterion. These requirements are automatically fulfilled by the theorem prover, since it is based on a typed λ-calculus. The tool uses a special variant of Hoare logic, which represents proofs explicitly within programs and thereby allows checking the correctness of programs (see chapter 10). Also, programs can be communicated, including their proofs. Hence, even the programming logic is safe and conforms to the De Bruijn criterion.

- The theorem prover should support several styles of reasoning. Cocktail seamlessly combines:

  - Backward reasoning: As is usual for interactive systems based on type-theory

- Forward reasoning: Supported through the use of a PTS ($\lambda P-$) with definitions. Definitions are inserted that represent the result of the forward reasoning step that is performed. Since definitions only occur within proof-terms and only due to forward reasoning, type checking does not become more difficult than type checking for $\lambda P-$ without definitions.
- Automated Theorem Proving: A Tableau based theorem prover is integrated in the system. Closed tableaux can be translated automatically into proof terms of the underlying PTS.
- Equational Reasoning: Leibniz equalities can be used for both expressions and formulas. Rewriting is supported by Leibniz substitutions and rewrite systems. Both can be used in forward and backward reasoning in a natural way.

16.1.3 The Designer's Requirements

The designer had a number of requirements related to the design and implementation of the tool. These requirements stated that the tool should be:

Adjustable: The PTS description at the base of the system can easily be changed. The programming logic is linked to the theorem prover logic only by the fake-statement. Hence, the programming logic and the proof logic can be altered independently.

Platform independent: this is achieved automatically by using Java as the implementation language. In practice, Java's speed is more than sufficient for the tool.

Modular: The tool is highly modular at all levels.

- Modules communicate the intended actions to each other by sending structured messages over a bus, allowing them to work together in a setting where they do not have to be aware of each others internal workings. Therefore, the editors, views, managers, automated theorem provers and other modules can be maintained and replaced independently of each other.
- The symbolic engine is a separate module, ensuring safety of the system. It is the only module that determines the formalisms

supported by the tool. Using a special Hoare logic allows the re-use of code from the type checker for the program checker (see chapter 15). This keeps the symbolic engine small. The logic can easily be extended by providing new classes to represent new syntactic structures. A type checker for the extended logic only has to deal with the new syntax and leave old syntax to the original type checker. This is already used in the tool: the program checker is an extension of the type checker of $\lambda P-$.

- The Tactic system contains support for backward, forward, automated and equational reasoning. Translating these tactics to smaller steps accepted by the symbolic engine is done by editors. Editors can also easily be extended with new tactics by inheritance of existing classes. This is already used in the system where, for example, the proof editor is an extension of the term editor.
- The user interface too is highly modular: displaying formulas, proofs and programs is done by using a displayer object that traverses their syntax trees (see chapter 15). To change the layout of formulas, proofs and programs, one merely has to change these displayer objects. Also, it is easily possible to have several displayer classes available and let the user choose her own display. New types of windows can easily be constructed by extending the view class. These windows can re-use the dedicated user interface components already available to render and interact with formulas, proofs, programs and tableaux.

Usable: Cocktail is remarkably easy to use through its advanced graphical user interface. As required, the Graphical User interface consists of a set of connected windows, displaying the global context (with axioms and proved theorems), proofs and programs. Through a flexible display system, proofs and programs are displayed like their pen and paper counterparts. Flexible formula display and parsing allow the usual infix notations of first order formulas and expressions. Interaction takes place directly with the displayed information. Buttons and menu's are only used for actions that cannot intuitively be linked to graphically displayed information. To ease selection of tactics, pop-up menu's are sensitive to the formula to which the tactic should be applied. Only relevant tactics are listed. Even when during rewriting of a formula several options exist, Cocktail simply prompts the user with several options from which she can select instead of resorting to complicated tree path notations to offer a selection mechanism.

16.2 Analysis of Cocktail's Shortcomings

Although Cocktail in its current state meets most of the requirements put forth by the programmer, the logician and the system's designer, there are a number limitations and improvements that we have to address. In this section, we will discuss which improvements for the tool are, in our judgment, needed in the future. We discuss them separately for each formal method supported by the tool to get a clear idea of the theory required for such improvements.

16.2.1 The Programming Part

Restricted Programming Language: In the near future extensions are needed for Arrays and Procedures. For these extension, the theory has already been discussed in chapter 8. Hence, they can be implemented in a couple of weeks. Later recursion, records, objects, pointers, modules/packages etc should be added as well. For these extensions the literature also offers numerous Hoare rules, but they should be selected with care in order not to destroy the safety of the system. Selecting and implementing those extensions would take somewhat more time and may be suitable topics for graduate projects.

Management of finished Programs: In the current version of the tool, completed programs cannot be re-used. Once we have procedures, this becomes easier. Every completed program can then be stored as a procedure and can be used within other programs as a single command, enabling the user to construct more complicated programs. Also, it should be possible to store and load programs and procedures independently of each other. Since all required theory is already in place (see chapter 8), this option could be added within about a month.

Theorem Proving within the Programming Editor: Rewriting specifications is not yet possible from within the program editor itself. A dedicated theorem prover interface for predicate calculus style of reasoning should be provided in addition to the natural deduction style theorem prover that is already implemented. Since such a theorem prover only requires rules for equational reasoning, it is easily implementable as an extension of the existing theorem prover: the required tactics are already available, one only has to adjust the user interface. (This is not difficult, but will take some time.) Also, this style of equational reasoning should be directly accessible from the programming window in the form of rewriting of pre- and postconditions.

Exporting algorithms: Procedures and programs should be export-able to (user defined) syntax other than that used by Cocktail (e.g. C, Java, Pascal). By providing specialized display algorithms this is technically easy and will take only little time. However, the user who supplies the syntax of the target language is responsible for the correctness of the translation. Functions declared within the program language context (Γ_1 in the Hoare logic of chapter 10) must exist in the target language and meet the specifications used by Cocktail.

16.2.2 The Interactive Theorem Prover

Management of large amounts of Theory: Facilities for creating and managing of packages of theory are needed. Re-ordering, renaming and deleting of theorems should be made possible. Also, it should be possible to replace an axiom (assumption) in a theory by a theorem and its proof. This allows hard theorems to be assumed and used as axioms first and be replaced by a proved theorem later (similar to COQ's 'grab axiom'). Although these are merely practical issues that have no theoretical impact, they are important for bookkeeping when developing large theories and programs. Implementing them, however, should be done very carefully, since re-ordering the context is not always possible (proofs are only allowed to depend on theorems and axioms occurring in the context *before* the proof itself). Hence, at least a few weeks are needed to add these facilities.

Displaying large proofs: For large proofs, it should be possible to hide parts that are already finished. Although the display algorithm easily allows this by annotating the root node of the sub-proof with a Hide-All value, the user is not authorized to do so yet. Implementing this should be a breeze.

Un-annotated proofs: Proofs should be annotated with formal comments (as proposed by Dijkstra and Feijen in their notation of predicate style proofs in [DF88]), written between the lines of the flagpole notation. This increases the readability of the proofs. Of course, this should be an option that can be switched on and off by the user, since it makes proofs substantially longer. This can be accommodated rather easily by adding new types of proof lines (namely comment lines) to the display algorithm. However, since all tactics then have to generate comments along with the proof-terms, implementation may take a bit of time.

16.2.3 The Automated Theorem Prover

Equational Automated Theorem Proving: The current version of the tool does not support equational reasoning during automatic proof construction. To support this, the ATP should be extended with rigid E-unification instead of the syntactic unification it uses now. Since tableaux have to be translated into λ-terms checkable by the tool for this to be safe, however, we first have to find a way to translate equational tableaux. As a result, we cannot yet estimate how long it would take to include this extension.

Cleaning up tableaux: Even without translation, tableaux can be unnecessary large. An algorithm should be used to reduce the size of the closed tableaux. This is possible by computing the subset of formulas at each node that is actually being used. Nodes that are inserted by rules that only produce unused formulas can then be removed. This might even lead to removal of entire branches of tableaux. A straightforward algorithm of this kind should be implementable in a few days. However, if one wants to use more sophisticated methods, the required effort may be larger.

Inefficient Translation of Tableaux: Translation of closed tableaux is inefficient at the time of this writing. The translation should be made an option, allowing the user to rely on the theorem prover's correctness. However, when the user is uncertain about a result she obtained, the tool should be able translate the automatically constructed proofs and check them for correctness afterwards. Implementing this is only a few days work. Since some users (for instance, programmers) may be less concerned about the safety and more about the performance, linking Cocktail to other ATPs, like Otter, Bliksem, Boyer-Moore etc should also be made an option. Of course, proofs of these systems cannot be translated to λ-terms, but they are usually capable of proving more theorems. Upgrading Cocktail's in-built ATP to the level of support that those systems offer would be a lot of work. Entire teams worked for years to build some of the other systems.

16.3 Conclusions

Cocktail has indeed become what we set out to build: A tool for deriving correct programs. It proves that it is possible to create a tool that meets the requirements formulated in part I. Cocktail combines a program editor with a theorem prover that allows the programmer to focus on the program and provide proofs at any moment she desires. The theorem prover allows the use of automated theorem proving whenever possible, but also has extensive support for interactive theorem proving and equational reasoning. It supports both backward and forward reasoning, including forward rewriting. The user can use familiar notations for proofs, programs and formulas. In fact, the notations are those used in pen and paper proofs.

Compared to other systems that support correct programming, Cocktail leaves the user with much more freedom in the order in which programs and proofs are constructed. If no full specification of the program is available (or the user does not wish to provide one), the program can easily be entered with a trivial specification. This specification can then be strengthened when the program is completed and the tool will compute the necessary proof-obligations. Hence, even though Cocktail is intended for hand in hand development of program and correctness proof, a process similar to generation of verification conditions (VCG) is supported. Cocktail provides a better integration of automated and interactive theorem proving than used in other systems. The ATP can be used at any given moment without compromising the safety of the system. Cocktail's advanced graphical user interface and its extensive support for forward reasoning allow the user to choose her own way to prove theorems. Interacting directly with the proofs as displayed appears to be intuitive and fast, but the use of the tool in an educational setting has yet to confirm these initial findings.

In order for the tool to become more practical and useful to a wider audience, a number of extensions are necessary. The programming language has to be extended to allow more advanced programs to be written and to re-use programs that have already been proved correct. The automated theorem prover must be enabled to use equational reasoning, but this requires the translation of equational tableaux which we have not yet investigated. Also, the tool must be able to deal with larger amounts of theory and programs. Most of these issues are mainly software engineering matters, that have little or no impact on the theoretical foundations of the tool. The required theories for Hoare logic are already available in literature and the flexible architecture of Cocktail will allow these extensions to be written relatively easy. However, since these extensions range over all parts of the tool, implementing all of them will require a few months. Still, we are planning to do so in the near future, probably with the help of graduate students.

Finally, we want to explain why the tool is called Cocktail and what the logo stands for. The name Cocktail was chosen because the tool implements a cocktail of formalisms, mixing PTSs, tableau methods and Hoare Logic. The logo, a cocktail glass with three spheres, represents the tool. Each sphere represents one formalism and the position in which they lay together represents the way in which these formalisms are related. The glass itself is a funnel, which combines the formalisms in the formal basis of the system, which is represented by the foot of the glass. The environment in which the glass is situated represents the participants in the project and all the additional bells and whistles of the tool.

Appendix A

A Small Demo Session

In this appendix, we will demonstrate Cocktail by deriving a program to compute Fibonacci numbers. For brevity, only part of the derivation is given in detail.

At the time of this writing, the implementation of the tactic system and the user-interface for the programming logic are not yet finished. As a result, not all tactics shown in this presentation are available in the actual tool. Instead, they were simulated by using smaller derivation steps that are already implemented. Since we expect that the mentioned tactics will be available by the time this thesis is defended, we have chosen to present them here as if they were available already. Hence, in the first version that will be made publicly available, there may be some minor differences with this presentation. All demonstrated features of the theorem prover, however, are fully functional as demonstrated.

After starting Cocktail, two windows appear: a control bar at the top of the screen and an empty theory-window (see figure A.1). Since we do not want to start building theory from scratch, we load some basic theory by clicking on the '`open`' button in the control bar and selecting the file '`prog.thm`'. After loading, these theories appear in the theory window.

To derive a program to compute Fibonacci numbers, we first have to add the Fibonacci function `fib` to the specification context. For this, we check the '`Only for specifications`' box and enter `fib(x:nat):nat` in the text-field at the bottom of the theory-window, thereby stating that `fib` is a function with one natural number as argument (the `x:nat`) that returns a

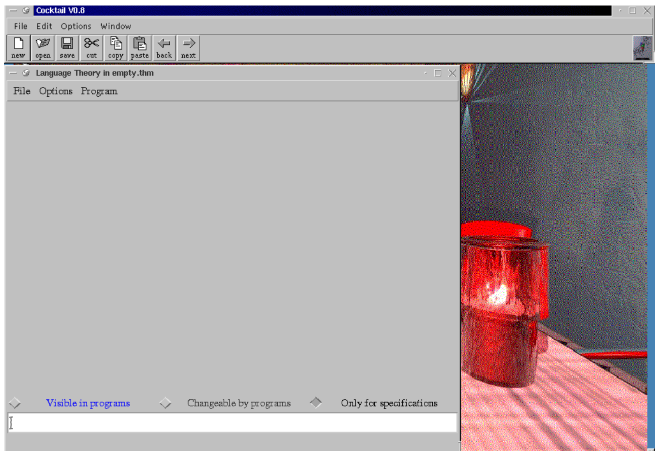


Figure A.1: Cocktail's control bar and the empty theory window.

natural number (the last `:nat`). Also, we enter the following axioms[1]:

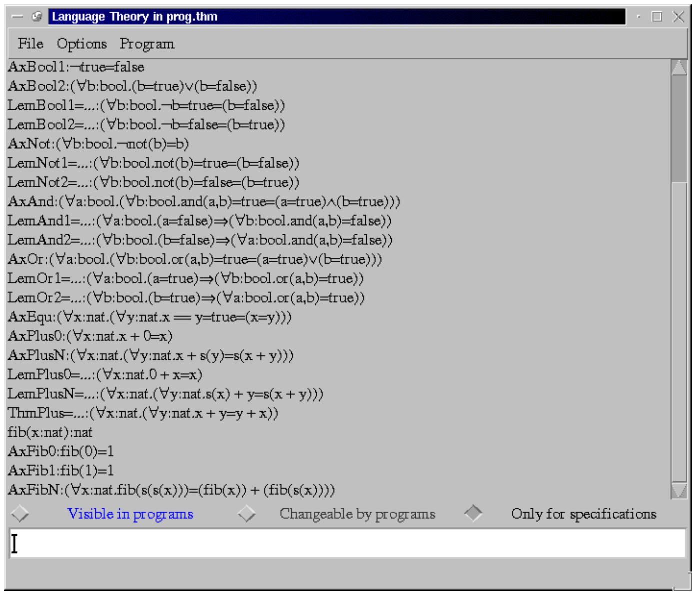
> `AxFib0:fib(0)=1`,
> `AxFib1:fib(s(0))=1` and
> `AxFibN:@x:nat.fib(s(s(x)))=fib(x)+fib(s(x))`.

The Fibonacci function and its specification are now available in the theory window (see figure A.2).

We can now specify the programming problem. First, we need to declare an initial constant `N()` and program variable `x`. We check the '`changeable by programs`' box and enter `N():nat` and `x:nat` in the theory window respectively. Next, we press the '`new`' button on the control bar to start a new program. A window appears in which we have to enter the new program's specification (see figure A.3). We enter the pre- and postcondition respectively as ~_ (not false, i.e. true) and `x=fib(N)`. Finally, we select '`done`' from the specification menu and the program window appears (see figure A.4).

We start programming with the tactic `replace constant`, available from the tactic menu. Since there is only one occurrence and only a single constant, we do not need to select a constant. When prompted for a variable

[1]When entering logical formulas, `@` means $\forall$, ~ means $\neg$ and _ means $\perp$. In future versions, buttons will be displayed below the text-field to enter special symbols.



```
AxBool1:¬true=false
AxBool2:(∀b:bool.(b=true)∨(b=false))
LemBool1=...:(∀b:bool.¬b=true=(b=false))
LemBool2=...:(∀b:bool.¬b=false=(b=true))
AxNot:(∀b:bool.¬not(b)=b)
LemNot1=...:(∀b:bool.not(b)=true=(b=false))
LemNot2=...:(∀b:bool.not(b)=false=(b=true))
AxAnd:(∀a:bool.(∀b:bool.and(a,b)=true=(a=true)∧(b=true)))
LemAnd1=...:(∀a:bool.(a=false)⇒(∀b:bool.and(a,b)=false))
LemAnd2=...:(∀b:bool.(b=false)⇒(∀a:bool.and(a,b)=false))
AxOr:(∀a:bool.(∀b:bool.or(a,b)=true=(a=true)∨(b=true)))
LemOr1=...:(∀a:bool.(a=true)⇒(∀b:bool.or(a,b)=true))
LemOr2=...:(∀b:bool.(b=true)⇒(∀a:bool.or(a,b)=true))
AxEqu:(∀x:nat.(∀y:nat.x == y=true=(x=y)))
AxPlus0:(∀x:nat.x + 0=x)
AxPlusN:(∀x:nat.(∀y:nat.x + s(y)=s(x + y)))
LemPlus0=...:(∀x:nat.0 + x=x)
LemPlusN=...:(∀x:nat.(∀y:nat.s(x) + y=s(x + y)))
ThmPlus=...:(∀x:nat.(∀y:nat.x + y=y + x))
fib(x:nat):nat
AxFib0:fib(0)=1
AxFib1:fib(1)=1
AxFibN:(∀x:nat.fib(s(s(x)))=(fib(x)) + (fib(s(x))))
```

Figure A.2: The basic theory for programs and the specification of the Fibonacci numbers are available in the theory window.

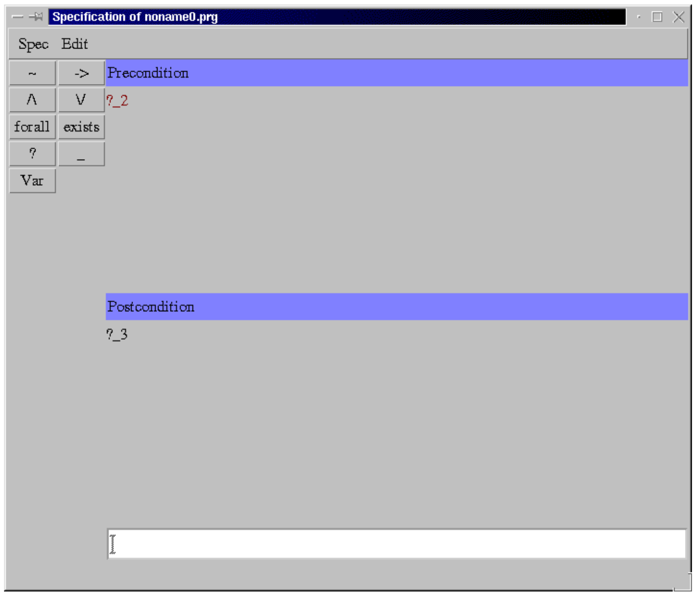
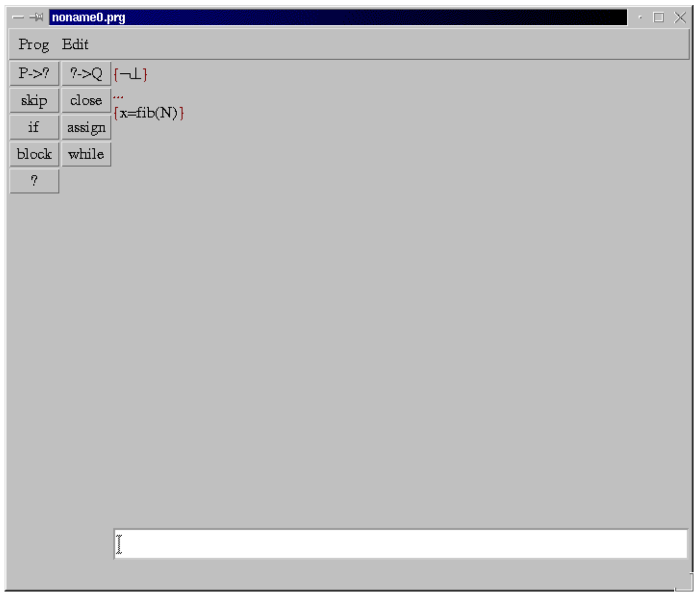


Figure A.3: A program specification window.



{¬⊥}
...
{x=fib(N)}

Figure A.4: The program window, showing the specification.

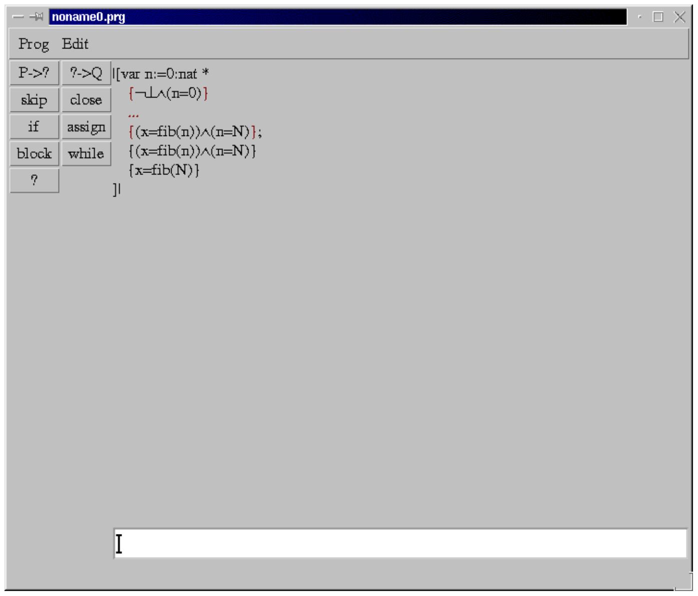
name and an initial value we enter `n` and `0` respectively. The program is now refined as shown in figure A.5: the new precondition is displayed as $\neg \perp \wedge n = 0$ and the postcondition is $x = \mathit{fib}(n) \wedge n = N$.

Next, we select the basic while-tactic, which prompts for an invariant and a boolean expression as guard. We enter `x=fib(n)` and `not(n==N)` respectively (`==` is defined as infix notation for `equ(x,y:nat):bool` and was loaded along with the basic theory). Also, we select the gap representing the body of the loop and enter `n:=s(n)` in the text-field at the bottom of the program window. The postcondition of the new gap becomes `x=fib(s(n))`. The result is shown in figure A.6. Among other things, we have a gap with a precondition displayed as $x = \mathit{fib}(n) \wedge \mathit{not}(n == N) = \mathit{false}$ and postcondition $x = \mathit{fib}(n) \wedge n = N$. This gap requires no more program refinement, but can be closed with a fake statement and a proof of

$$(x = \mathit{fib}(n) \wedge \mathit{not}(n == N) = \mathit{false}) \Rightarrow (x = \mathit{fib}(n) \wedge n = N)$$

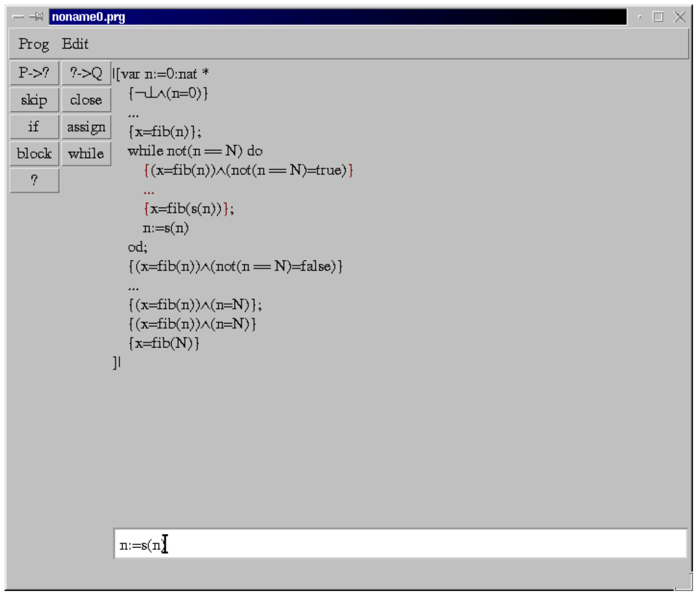
To construct this proof, we select the gap by clicking on it and select the '`close`' tactic. As a result, a theorem prover window appears.

The theorem prover window (see figure A.7) shows the proof obligation we



```
|[var n:=0:nat *
   {¬⊥∧(n=0)}
   ...
   {(x=fib(n))∧(n=N)};
   {(x=fib(n))∧(n=N)}
   {x=fib(N)}
]|
```

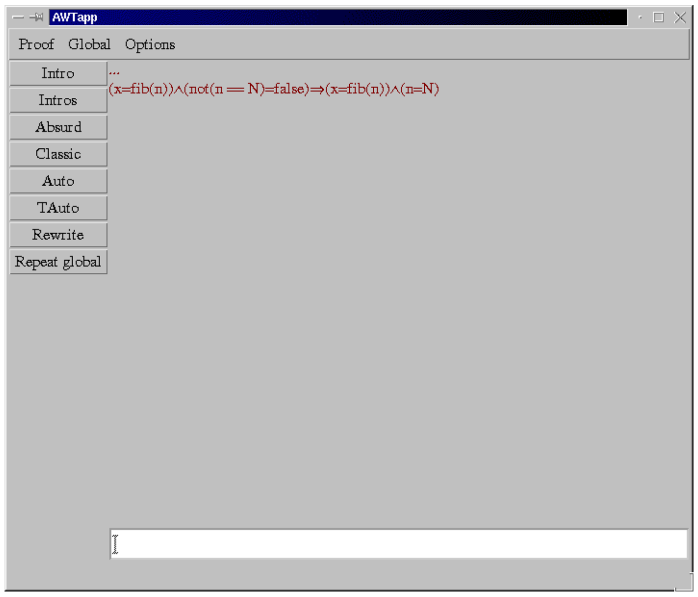
Figure A.5: The constant `N` has been replaced by a variable named `n`.



```
|[var n:=0:nat *
   {¬⊥∧(n=0)}
   ...
   {x=fib(n)};
   while not(n == N) do
      {(x=fib(n))∧(not(n == N)=true)}
      ...
      {x=fib(s(n))};
      n:=s(n)
   od;
   {(x=fib(n))∧(not(n == N)=false)}
   ...
   {(x=fib(n))∧(n=N)};
   {(x=fib(n))∧(n=N)}
   {x=fib(N)}
]|
```

```
n:=s(n
```

Figure A.6: The initial loop. We now have three gaps (proof obligations) in the program.



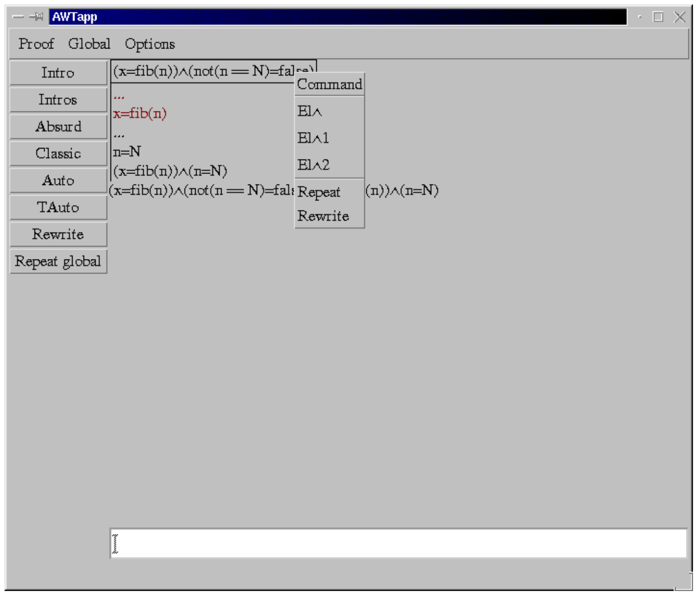
```
...
(x=fib(n))∧(not(n == N)=false)⇒(x=fib(n))∧(n=N)
```

Figure A.7: The theorem prover window, showing a proof obligation.

have to fulfill. Since we are dealing with an implication, we use the button labeled with '`intro`' to raise a flag. The new goal is to prove $x = \mathit{fib}(n) \wedge n = N$, given the assumption $x = \mathit{fib}(n) \wedge \mathit{not}(n == N) = \mathit{false}$. To split the new goal into two separate parts, we use the right mouse button to click on the goal. Right-clicking on the assumption $x = \mathit{fib}(n) \wedge \mathit{not}(n == N) = \mathit{false}$ shows a number of options for forward reasoning (see figure A.8).

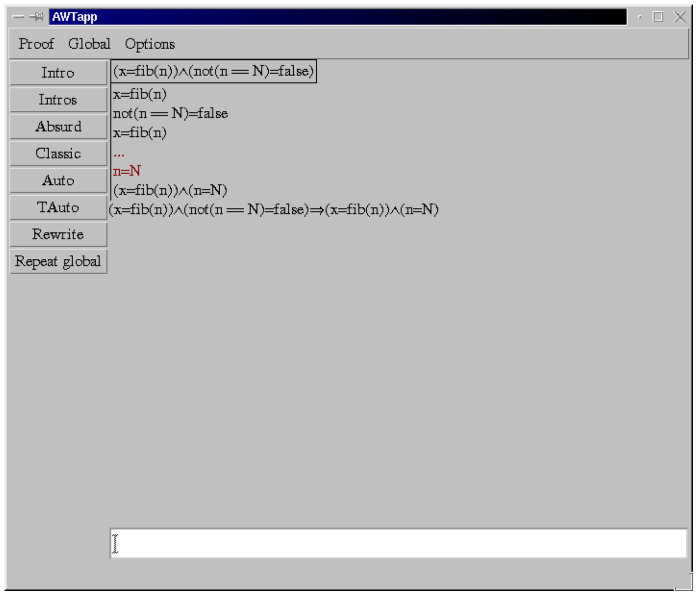
We select the option '`El`∧' to obtain both parts of the assumption as separate items. To resolve the goal $x = \mathit{fib}(n)$, first select it and then click on the new item with the same statement. We are left with one goal (see figure A.9): proving $n = N$.

Since we already have $\mathit{not}(n == N) = \mathit{false}$, we can use the theorems in the theory window to rewrite this formula into $n = N$ using Leibniz-equalities. Rewriting a boolean to a proposition, however, is often needed in program derivations. Therefore, we do not use individual Leibniz-equalities, but use the rewrite system '`bool2prop`', loaded with the initial theory. To do this, right click the item $\mathit{not}(n == N) = \mathit{false}$ and select the '`rewrite`' option. Select '`bool2prop`' at the top of the displayed list of rewrite-rules (see figure A.10) and press either the '`left`' or the '`right`' button (Since



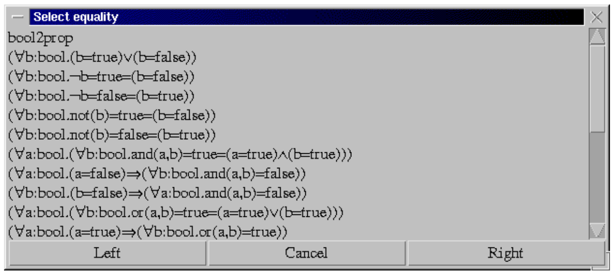
```
(x=fib(n))∧(not(n == N)=fal
...
x=fib(n)
...
n=N
(x=fib(n))∧(n=N)
(x=fib(n))∧(not(n == N)=fal          (n))∧(n=N)
```

Figure A.8: A natural deduction proof with two subgoals in flagpole notation. Tactics for forward reasoning are displayed in a pop-up menu.



```
(x=fib(n))∧(not(n == N)=false)
x=fib(n)
not(n == N)=false
x=fib(n)
...
n=N
(x=fib(n))∧(n=N)
(x=fib(n))∧(not(n == N)=false)⇒(x=fib(n))∧(n=N)
```

Figure A.9: The first goal has been resolved.



```
bool2prop
(∀b:bool.(b=true)∨(b=false))
(∀b:bool.¬b=true=(b=false))
(∀b:bool.¬b=false=(b=true))
(∀b:bool.not(b)=true=(b=false))
(∀b:bool.not(b)=false=(b=true))
(∀a:bool.(∀b:bool.and(a,b)=true=(a=true)∧(b=true)))
(∀a:bool.(a=false)⇒(∀b:bool.and(a,b)=false))
(∀b:bool.(b=false)⇒(∀a:bool.and(a,b)=false))
(∀a:bool.(∀b:bool.or(a,b)=true=(a=true)∨(b=true)))
(∀a:bool.(a=true)⇒(∀b:bool.or(a,b)=true))
```

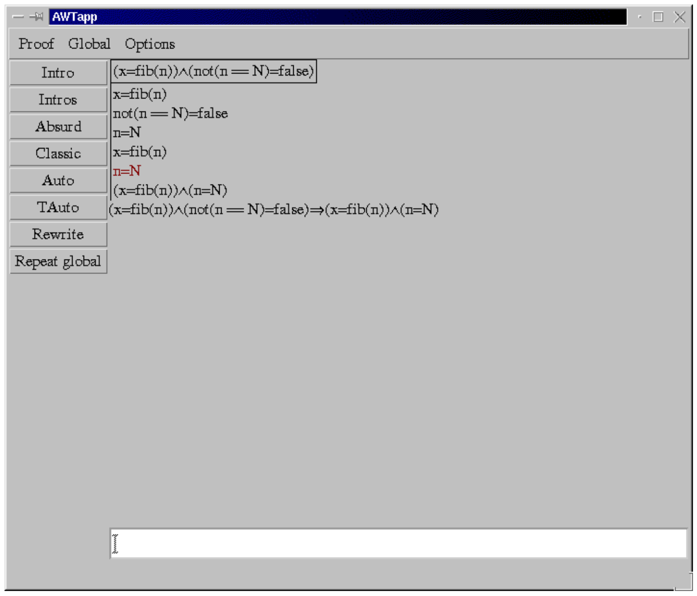
Figure A.10: A list of possible rewrite rules.

the rewrite-directions are specified within the rewrite-system, there is no difference between left- and right rewriting).

The result of the rewriting is added as an item to the context, hence we now have $n = N$ available in our proof. Clicking on this new item closes the final gap and yields the final proof (see figure A.11). We now close the window by selecting '`done`' from the proof-menu (the result is type-checked before the window closes). The gap in the program is now closed (see figure A.12).

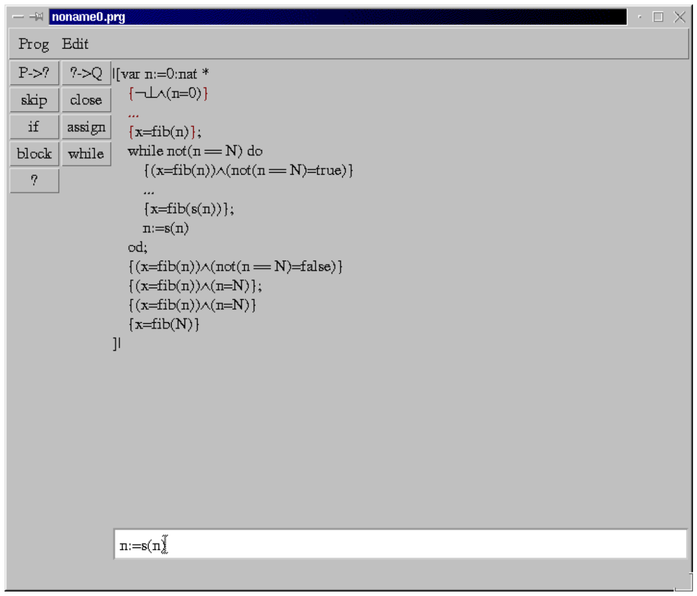
After a few more refinements and proofs, our program looks as shown in figure A.13. At this point, we need to strengthen the invariant by adding a conjunct $y = \mathit{fib}(s(n))$ to it, where y is a fresh-variable. This is done in two steps: (1) Selecting the loop and adding a fresh variable, yielding the situation in figure A.14. (2) Strengthening the invariant with $y = \mathit{fib}(s(n))$, yielding figure A.15.

Some more refinements are needed to obtain the final program (see figure A.16). At this point, we merely have to provide the proofs required to close the gaps. Figure A.17 shows the completed program, which has been checked by Cocktail.



```
(x=fib(n))∧(not(n == N)=false)
x=fib(n)
not(n == N)=false
n=N
x=fib(n)
n=N
(x=fib(n))∧(n=N)
(x=fib(n))∧(not(n == N)=false)⇒(x=fib(n))∧(n=N)
```

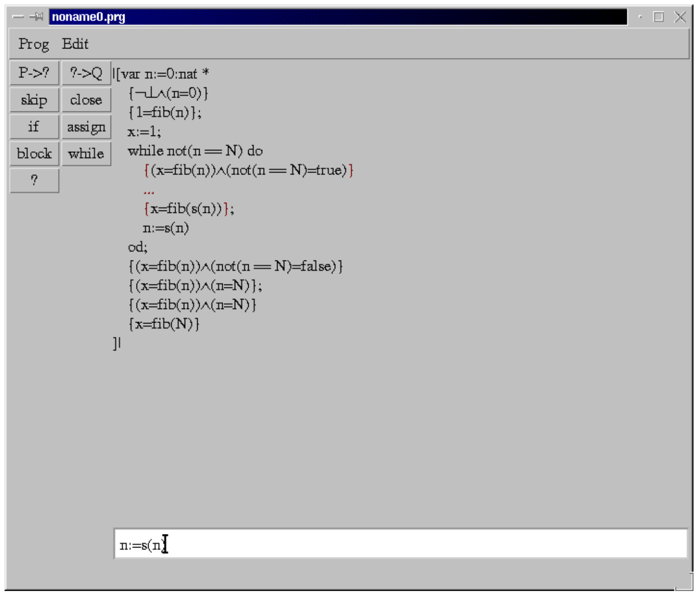
Figure A.11: The completed proof, which will be type-checked before it is used.



```
|[var n:=0:nat *
  {¬⊥∧(n=0)}
  ...
  {x=fib(n)};
  while not(n == N) do
    {(x=fib(n))∧(not(n == N)=true)}
    ...
    {x=fib(s(n))};
    n:=s(n)
  od;
  {(x=fib(n))∧(not(n == N)=false)}
  {(x=fib(n))∧(n=N)};
  {(x=fib(n))∧(n=N)}
  {x=fib(N)}
]|
```

```
n:=s(n)
```

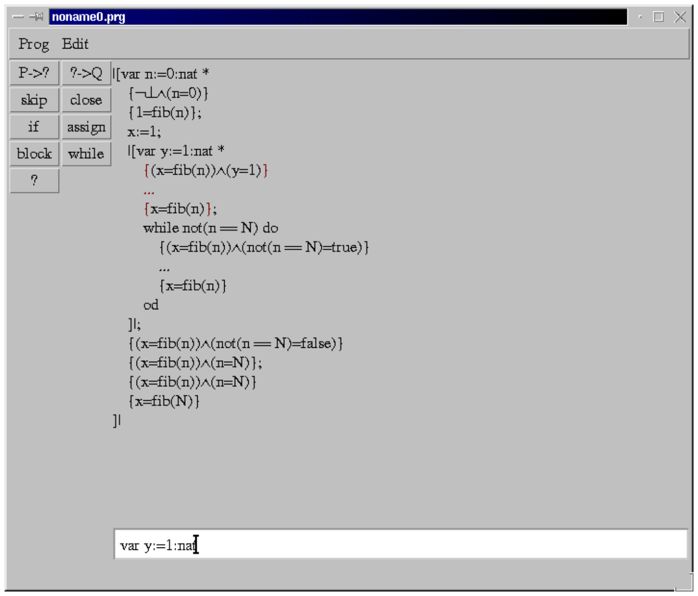
Figure A.12: The same program as in figure A.6, but with one gap closed.



```
|[var n:=0:nat *
  {¬⊥∧(n=0)}
  {1=fib(n)};
  x:=1;
  while not(n == N) do
    {(x=fib(n))∧(not(n == N)=true)}
    ...
    {x=fib(s(n))};
    n:=s(n)
  od;
  {(x=fib(n))∧(not(n == N)=false)}
  {(x=fib(n))∧(n=N)};
  {(x=fib(n))∧(n=N)}
  {x=fib(N)}
]|
```

```
n:=s(n)
```

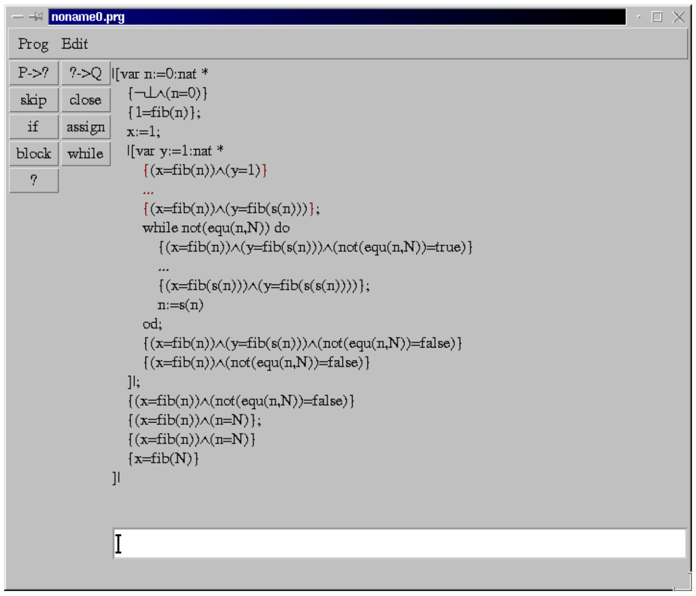
Figure A.13: Some more refinements done.



```
|[var n:=0:nat *
  {¬⊥∧(n=0)}
  {1=fib(n)};
  x:=1;
  |[var y:=1:nat *
    {(x=fib(n))∧(y=1)}
    ...
    {x=fib(n)};
    while not(n == N) do
      {(x=fib(n))∧(not(n == N)=true)}
      ...
      {x=fib(n)}
    od
  ]|;
  {(x=fib(n))∧(not(n == N)=false)}
  {(x=fib(n))∧(n=N)};
  {(x=fib(n))∧(n=N)}
  {x=fib(N)}
]|
```

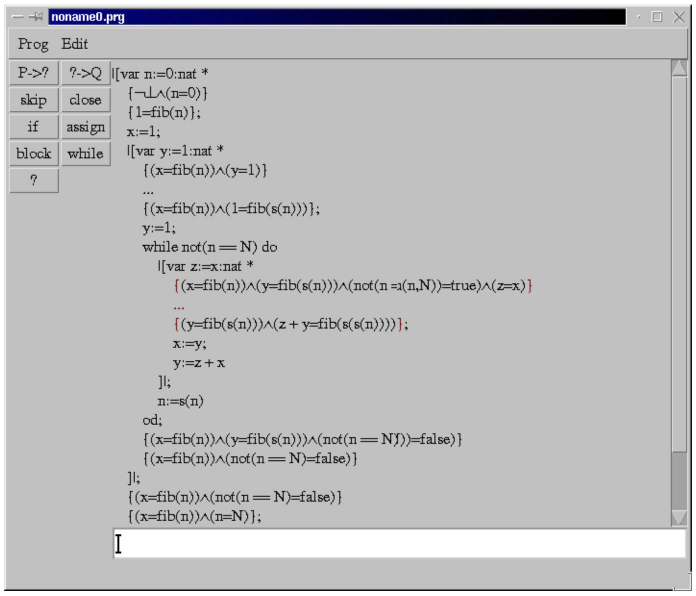
```
var y:=1:nat
```

Figure A.14: A fresh variable named y was added.



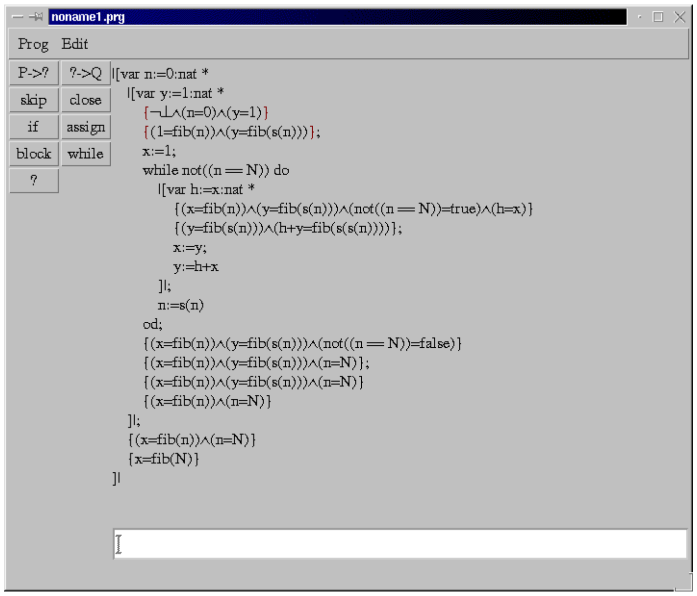
```
|[var n:=0:nat *
  {¬⊥∧(n=0)}
  {1=fib(n)};
  x:=1;
  |[var y:=1:nat *
    {(x=fib(n))∧(y=1)}
    ...
    {(x=fib(n))∧(y=fib(s(n)))};
    while not(equ(n,N)) do
      {(x=fib(n))∧(y=fib(s(n)))∧(not(equ(n,N))=true)}
      ...
      {(x=fib(s(n)))∧(y=fib(s(s(n))))};
      n:=s(n)
    od;
    {(x=fib(n))∧(y=fib(s(n)))∧(not(equ(n,N))=false)}
    {(x=fib(n))∧(not(equ(n,N))=false)}
  ]|;
  {(x=fib(n))∧(not(equ(n,N))=false)}
  {(x=fib(n))∧(n=N)};
  {(x=fib(n))∧(n=N)}
  {x=fib(N)}
]|
```

Figure A.15: After strengthening the invariant, many sub-specifications have changed.



```
|[var n:=0:nat *
  {¬⊥∧(n=0)}
  {1=fib(n)};
  x:=1;
  |[var y:=1:nat *
    {(x=fib(n))∧(y=1)}
    ...
    {(x=fib(n))∧(1=fib(s(n)))};
    y:=1;
    while not(n == N) do
      |[var z:=x:nat *
        {(x=fib(n))∧(y=fib(s(n)))∧(not(n =ı(n,N))=true)∧(z=x)}
        ...
        {(y=fib(s(n)))∧(z + y=fib(s(s(n))))};
        x:=y;
        y:=z + x
      ]|;
      n:=s(n)
    od;
    {(x=fib(n))∧(y=fib(s(n)))∧(not(n == N}))=false)}
    {(x=fib(n))∧(not(n == N)=false)}
  ]|;
  {(x=fib(n))∧(not(n == N)=false)}
  {(x=fib(n))∧(n=N)};
```

Figure A.16: The final program, but with some unresolved proof-obligations.



```
|[var n:=0:nat *
   |[var y:=1:nat *
      {¬⊥∧(n=0)∧(y=1)}
      {(1=fib(n))∧(y=fib(s(n)))};
      x:=1;
      while not((n == N)) do
         |[var h:=x:nat *
            {(x=fib(n))∧(y=fib(s(n)))∧(not((n == N))=true)∧(h=x)}
            {(y=fib(s(n)))∧(h+y=fib(s(s(n))))};
            x:=y;
            y:=h+x
         ]|;
         n:=s(n)
      od;
      {(x=fib(n))∧(y=fib(s(n)))∧(not((n == N))=false)}
      {(x=fib(n))∧(y=fib(s(n)))∧(n=N)};
      {(x=fib(n))∧(y=fib(s(n)))∧(n=N)}
      {(x=fib(n))∧(n=N)}
   ]|;
   {(x=fib(n))∧(n=N)}
   {x=fib(N)}
]|
```

Figure A.17: The completed program, derived with and verified by Cocktail.

Appendix B

Factsheet

Since Cocktail is still under construction, the following facts about its implementation are preliminary. All data were measured at Friday, 13 October 2000.

B.1 Implementation

To install Cocktail, one needs its binary files and a Java Virtual Machine (JVM). Virtual machines for Java are freely available for many platforms through the website of SUN microsystems (`http://java.sun.com/`). The size of the JVM depends on the platform.

About the java implementation of Cocktail, the following facts were measured:

Number of packages:	8
Number of files:	197
Number of lines:	15738
Number of methods:	2520
Number of classes:	234
Number of interfaces:	47

The file sizes are measured for each package separately in kilobyte:

Package	total	source	binary
auto	108	56	56
bus	168	88	84
editors	420	232	192
hoare	248	128	124
misc	292	128	168
terms	496	244	256
views	492	200	304
main	184	160	168
total	2404	1236	1352

B.2 Javadoc documentation

Java supports automatic extraction of API documentation in HTML format from the source code. Also, additional comments can be added by the programmer which will automatically be added to the API documentation. Facts about the API documentation of Cocktail are:

Total size (kByte):	2444
Number of files:	204
Number of lines:	43606

B.3 Availability

Cocktail will be available in due time through its website at

`http://www.win.tue.nl/~michaelf/cocktail.html`

At this site, documentation, tutorials, references and all other information will also be made available.

Bibliography

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey - part I. *ACM Transactions on Programming Languages and Systems*, 3(4):432–483, October 1981.

[Bar92] H.P. Barendregt. Lambda calculi with types. In S Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*, pages 118–310. Oxford Science Publications, 1992.

[BHN00] Marc Bezem, Dimitry Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. In D McAllester, editor, *17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Computer Science*, Pittsburgh, USA, June 17–20 2000. Springer.

[BJMP93] L.S. van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for pure type systems. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Propositions: International Workshop TYPES'93*, volume 806 of *LNCS*, pages 19–61, Nijmegen, May 1993. Springer-Verlag 1994.

[BKT94] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 1994.

[BM88] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[Bor00] Richard Bornat. Proving pointer programs in Hoare logic. In *Proceeding of the fifth international conference on the Math-*

ematics in Program Construction 2000. Springer, 2000. To appear.

[BS] Richard Bornat and Bernard Sufrin. The JAPE homepage. `http: //users.comlab.ox.ac.uk/ bernard.sufrin/ jape.html`

[BS98] Wolfgang Bibel and Peter H. Schmitt, editors. *Automated Deduction - A Basis for Applications*, volume I : Foundations - Calculi and Methods of *Applied Logic Series.* Kluwer Academic Publishers, 1998.

[Bur72] R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence 7*, pages 23–50, 1972.

[CAB86] R.L. Constable, S.F. Allen, and H.M. Bromley. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, 1986.

[Chr93] Heine Christensen. Synthesis of programs from logic specifications using programming methodology. *Structured Programming*, 14:173–186, 1993.

[Coq97] Coq. The Coq proof assistant. In *URL:* `http: //coq.inria.fr/`, 1997.

[DF88] Edsger W. Dijkstra and W.H.J. Feijen. *A Method of Programming.* Addison-Wesley, 1988.

[DGHP99] M D'Agostino, D Gabbay, R Hähnle, and J Posegga, editors. *Handbook of Tableau Methods.* Kluwer Academic Publishers, 1999.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.

[dK95] Eric de Kogel. *Equational Proofs in Tableaux and Logic Programming.* PhD thesis, Tilburg University, 1995.

[dN95] Hans de Nivelle. *Ordering Refinements of Resolution.* PhD thesis, Delft University of Technology, 1995.

[dS93] H.C.M. de Swart. *LOGIC; Mathematics, Language, Computer Science and Philosophy*, volume I. Peter Lang, Frankfurt, 1993.

[Fit52] Frederic Brenton Fitch. *Symbolic Logic, an Introduction.* The Ronald Press, New York, 1952.

[Fla97] David Flanagan. *Java in a nutshell.* O'Reilly, 1997.

[Fra97] Michael Franssen. Tools for the construction of correct programs: an overview. Technical Report Report 97-06, Eindhoven University of Technology, 1997.

[Fra00] Michael Franssen. Embedding first-order tableaux into a pure type system. In Didier Galmiche, editor, *Electronic Notes in Theoretical Computer Science*, volume 17. Elsevier Science Publishers, 2000.

[FVW00] Michael Franssen, Remco Veltkamp, and Wieger Wesselink. Efficient evaluation of triangular B-splines. *Computer Aided Geometric Design*, 17:863–877, 2000.

[GGH98] Carl A. Gunter, Elsa L. Gunter, and P. Homeier. Sunrise: A Verified Verification Condition Generator. In *URL:* `http://www.cis.upenn.edu/ ~hol/sunrise/`, 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable object-oriented software.* Professional Computing Series. Addison-Wesley, 1995.

[GL80] David Gries and G. Levin. Assignment and procedure call proof rules. *ACM TOPLAS*, 2:564–579, 1980.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, Cambridge, 1993.

[Gor75] G.A. Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, University of Toronto, Department of Computer Science, 1975.

[Gri81] David Gries. *The Science of Programming.* Springer, 1981.

[HBG94] Reiner Hähnle, Bernhard Beckert, and Stefan Gerberding. The many-valued tableau-based theorem prover 3tap. Technical Report 30/94, Institute for Logic, Complexity and Deduction Systems, 1994.

[Hei92] Maritta Heisel. Formalizing and implementing Gries' program development method in dynamic logic. *Science of Computer Programming*, 18:107–137, 1992.

[HLS97] F.W. von Henke, M. Luther, and M. Strecker. Typelab: An environment for modular program development. In Michel Bidoit and Max Dauchet, editors, *Tapsoft'97 Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 851–854. Springer, April 1997.

[HM96] Peter V. Homeier and David F. Martin. Mechanical verification of mutually recursive procedures. In M.A. McRobbie and J.K. Slaney, editors, *Automated Deduction CADE-13*, Lecture Notes in Artificial Intelligence, pages 201–215. Springer, July/August 1996.

[HMP93] Robert Harper, Furio Monsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Januari 1993.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[Hue89] Gérard Huet. *A Perspective in Theoretical Computer Science*, volume 16 of *World Scientific Series in Computer Science*, chapter The Constructive Engine. World Scientific, 1989.

[Kal90] Anne Kaldewaij. *Programming: the derivation of algorithms.* Prentice-Hall international series in Computer Science. Prentice Hall, 1990.

[Laa97] T. Laan. *The Evolution of Type Theory in Logic and Mathematics.* PhD thesis, Eindhoven University of Technology, 1997.

[LEG97] LEGO. The LEGO proof assistant. In *URL:* `http: //www.dcs.ed.ac.uk/ home/lego/`, 1997.

[LF00] Twan Laan and Michael Franssen. Embedding first-order logic in a pure type system with parameters. *Journal of Logic and Computation*, 2000. Accepted for publication.

[McC94] William W. McCune. Otter 3.0 reference manual and guide. Technical report, Argonne National Laboratory, Januari 1994.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In Jonathan L. Cohen, Jerry Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski, editors, *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North-Holland.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2), April 1982.

[Ned77] Rob P. Nederpelt. Presentation of natural deduction. In *Set theory, foundations of mathematics*, volume 2 of *Nouv-série tome*, pages 115–125, Beograd, 1977. Recueil des travaux de l'Institut Mathématique.

[NGdV94] Rob Nederpelt, Herman Geuvers, and Roel de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and The Foundations of Mathematics*. Elsevier, 1994.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* Wiley Professional Computing. Wiley, 1992.

[Oph92] W.J. Ophelders. *Automated Theorem Proving Based Upon a Tableau-Method With Unification Under Restrictions: Theory, Implementation and Empirical Results.* PhD thesis, Tilburg University, 1992.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. CADE, Springer Verlag, 1992.

[Pau94] Lawrence C. Paulson. *Isabelle: a generic theorem prover.* Springer, Berlin, 1994.

[Pfe91] Frank Pfenning. *Logical Frameworks*, chapter Logic Programming in the LF Logical Framework, pages 149–181. Cambridge University Press, 1991.

[PM89] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. In *Sixteenth Annual ACM symposium on Principles of Programming Languages*, pages 89–104, Austin, Texas, Januari 1989. ACM, ACM press.

[Pol94] Erik Poll. *A Programming Logic Based on Type Theory.* PhD thesis, Eindhoven University of Technology, 1994.

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

[Smu68] Raymond M. Smullyan. *First-Order Logic.* Ergebnisse der Mathematik und Ihrer Grenzgebiete, Band 43. Springer, 1968.

[TBK92] Laurent Théry, Yves Bertot, and Gilles Kahn. Real theorem provers deserve real user-interfaces. Technical Report 1684, INRIA Sophia-Antipolis, Mai 1992.

[Ter89] Jan Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Technical report, Department of Computer Science, University of Nijmegen, 1989.

[vdBvDK96] Mark van den Brand, Arie van Deursen, and Paul Klint. Industrial applications of ASF+SDF. Technical Report CS-R 9622, Centrum voor Wiskunde en Informatica, Amsterdam, 1996.

[vW94] Joakim von Wright. Program refinement by theorem prover. In David Till, editor, *6th Refinement Workshop*, pages 121–150. Springer, 1994.

[Zwa97] Jan Zwanenburg. The Yarrow home page. In *URL:* `http://www.cs.kun.nl/~janz/yarrow/`, 1997.

[Zwa98] Jan Zwanenburg. The proof-assistent yarrow. Computing Science Reports 98–11, Eindhoven University of Technology, 1998.

[Zwa99] Jan Zwanenburg. *Object-Oriented Concepts and Proof Rules: Formalization in Type Theory and Implementation in Yarrow.* PhD thesis, Eindhoven University of Technology, 1999.

Summary

The goal of this thesis is to obtain machine assistance for the Dijkstra/Hoare calculus. This calculus has already proved itself valuable as a pen-and-paper method to derive correct programs, but when program derivations and correctness proofs are carried out manually, it is prone to human mistakes. By offering machine assistance for the method, we get help when deriving programs, gain precision, avoid errors and can verify the result once the program and proof are complete. This machine assistance has been designed in the form of Cocktail.

Cocktail is an interactive tool to derive correct programs from their specifications using Dijkstra/Hoare calculus. In this calculus, programs are refined, guided by their specification, leading to a number of proof obligations, which have to be resolved. The tool allows the user to interactively construct an entire program and a complete proof of its correctness. For the construction of correctness proofs, Cocktail offers a proof-editor for first-order logic, which is partially automated by a tableau based theorem prover.

The tool is initially designed to support courses for first and second year students of computer science at Eindhoven University of Technology. In the future, we want to extend the tool in order to support more advanced courses and perhaps industrial-type programs.

The main characteristics of Cocktail are:

Program targeted: The tool allows the user to focus on the program, rather than on its correctness proof.

Full support: Programs and proofs can interactively be constructed by the user entirely within the tool.

Natural: Programs are developed using a well-established, natural method

which promotes a standardized design method. The same holds for theorem proving: several styles of reasoning are supported in a uniform way (backward reasoning, forward reasoning, automated theorem proving and rewriting). Moreover, the tool is equipped with an advanced graphical user interface that allows the user to keep a good overview of the program and its correctness proofs during the entire process.

Safe: Once the program and its correctness proof are complete, they are automatically checked for correctness by a simple algorithm. This check will only fail if errors exist in the implementation of the tool: the construction method implies that the program should have been correct.

Flexible: The tool is easy to maintain, adjust and extend due to its modular, object oriented design.

To accomplish this, the following theoretical work was done:

- A Pure Type System (PTS) was designed to accurately model first-order logic. Using the PTS framework makes it easier to extend the logic of the tool in the future.
- An algorithm to translate closed tableaux into λ-terms of the prementioned PTS was designed. This enables the tool to use meaningful automated theorem proving in the logic.
- A special kind of Hoare logic was designed. Hoare logic enables the tool to focus on the program and allows it to perform refinement steps. The Hoare logic used is special by the way it is linked to the theorem prover's logic and by its explicit use of contexts and proofs. These novelties, however, enable the verification of completed programs and ensure that even the Hoare logic conforms to the De Bruijn criterion (this criterion states that proofs should be communicable to other systems).

The implementation of the formal systems, in the form of Cocktail, resulted in a coherent tool for deriving correct programs focused at the programming process. The automated theorem prover should be extended to deal with equations. Cocktail's formal foundation, including the automated theorem prover and the Hoare logic, is safe: all constructed programs and proofs are checked by the tool and conform to the De Bruijn criterion. The tool offers a graphical user interface that is remarkably easy to use, since it allows the user to interact directly with the displayed proofs and programs instead of resorting to a command line interface. Also, the tool can easily be extended. Although Cocktail is currently not suitable to be used to derive large programs, it certainly has the required potential to do so in the future.

Samenvatting

In dit proefschrift ontwikkelen we een programma dat ondersteuning biedt voor de Dijkstra/Hoare-calculus. Deze calculus heeft reeds bewezen nuttig te zijn voor het op papier afleiden van correcte programma's. Doordat echter alle afleidingen en bewijzen tot dusver met de hand werden gemaakt, is de methode gevoelig voor menselijke vergissingen. Door de methode met een programma te ondersteunen krijgen we hulp bij het afleiden, werken we nauwkeuriger, voorkomen we fouten en kunnen we het resultaat controleren als het programma en het bewijs af zijn. De ontwikkelomgeving die hiervoor is ontwikkeld heeft de naam Cocktail gekregen.

Cocktail is een interactief systeem waarmee programma's kunnen worden afgeleid uit hun specificaties door gebruik te maken van Dijkstra/Hoare-calculus. Hierbij worden programma's, op basis van hun specificatie, stapsgewijs verfijnd, hetgeen leidt tot een aantal bewijsverplichtingen waaraan moet worden voldaan. De ontwikkelomgeving biedt de mogelijkheid voor het maken van het volledige programma en het volledige correctheidsbewijs. Voor het maken van het correctheidsbewijs biedt Cocktail een bewijseditor voor eerste orde logica, die deels geautomatiseerd is door middel van een tableau-bewijzer.

De omgeving is in eerste instantie ontworpen om eerste- en tweedejaars studenten van de Technische Universiteit Eindhoven te ondersteunen bij het programmeren. In de toekomst willen we de ontwikkelomgeving uitbreiden voor ondersteuning van hogerejaars cursussen en wellicht industriële programma's.

De belangrijkste kenmerken van Cocktail zijn:

Programmagerichtheid: De omgeving is gericht op het programma en niet zozeer op het correctheidsbewijs ervan.

Volledige ondersteuning: Zowel het programma als alle benodigde bewijzen kunnen volledig met Cocktail worden gemaakt.

Natuurlijk: Programma's worden ontwikkeld met een natuurlijke methode die zich al bewezen heeft en die een standaard programmeeraanpak bevordert. Hetzelfde geldt voor de bewijzen: verschillende manieren voor het maken van logische bewijzen worden op een uniforme wijze ondersteund (achterwaarts redeneren, voorwaarts redeneren, automatisch stellingen bewijzen en herschrijven). Bovendien biedt de omgeving een geavanceerd grafisch user-interface waardoor de gebruiker gedurende het hele proces een goed overzicht behoudt over het programma en het bewijs ervan.

Veilig: Als het programma en het bewijs eenmaal af zijn, worden ze nogmaals gecontroleerd met een eenvoudig algoritme. Als hierbij fouten worden gevonden, wijst dat op fouten in Cocktail: wegens de gehanteerde ontwikkelmethode hadden de resultaten foutloos moeten zijn.

Flexibel: De ontwikkelomgeving is gemakkelijk te onderhouden, aan te passen en uit te breiden dankzij haar modulaire, object georiënteerde ontwerp.

Om dit voor elkaar te krijgen is het volgende theoretische werk verricht:

- Er is een puur typesysteem (PTS) ontworpen dat precies overeenkomt met eerste orde logica. Het gebruiken van het PTS-raamwerk maakt het makkelijker om de logica in de toekomst nog uit te breiden.

- Er is een algoritme geschreven om gesloten tableaus te vertalen naar λ-termen van het voornoemde PTS. Hierdoor kan de bewijslogica op een zinvolle manier worden geautomatiseerd.

- We hebben een speciaal soort Hoare-logica ontwikkeld. De Hoare-logica is nodig om de omgeving programmagericht te houden en om verfijningsstappen van het programma te kunnen uitvoeren. De ontwikkelde Hoare-logica is afwijkend door de manier waarop deze gekoppeld is aan de bewijslogica en doordat contexten en bewijzen expliciet zijn gemaakt. Deze kenmerken zorgen ervoor dat afgeleide programma's op correctheid gecontroleerd kunnen worden en dat zelfs de Hoare-logica aan het De Bruijn-criterium voldoet (dit criterium eist dat bewijzen naar andere systemen gecommuniceerd kunnen worden).

De implementatie van de formele systemen in de vorm van Cocktail heeft geleid tot een coherent stuk gereedschap voor het afleiden van correcte programma's dat gericht is op de programmeertaak. De automatische stellingenbewijzer moet nog uitgebreid worden om met gelijkheden te kunnen werken. Cocktails formele grondslagen, inclusief de automatische stellingenbewijzer en de Hoare-logica, zijn veilig: de omgeving verifieert alle gemaakte programma's en bewijzen en voldoet aan het De Bruijn-criterium. Cocktail biedt een opmerkelijk gemakkelijk te gebruiken user interface, doordat het de gebruiker rechtstreeks de afgebeelde programma's en bewijzen laat manipuleren in plaats van gebruik te maken van het gebruikelijke command-line interface. Bovendien is de ontwikkelomgeving gemakkelijk uit te breiden. Hoewel Cocktail momenteel nog niet geschikt is voor het afleiden van grote programma's, heeft het wel het potentieel om dat in de toekomst te kunnen.

Curriculum Vitae

In 1971 werd ik op 11 maart geboren in Heerlen. In 1983 begon ik mijn opleiding aan de MAVO Bronsheim te Brunssum (toen nog MAVO Sancta Maria). Ik kwam toendertijd voor het eerst met computers in aanraking en had vooral veel interesse in programmeren. In 1987 begon ik aan de opleiding electronica aan de MTS Heerlen. Het stagejaar op de MTS sloeg ik over en in 1990 zette ik mijn electronica-opleiding voort op de HTS in Heerlen. Na het behalen van mijn propedeuse vertrok ik in 1991 naar de Technische Universiteit Eindhoven (TUE) voor een studie Informatica. Naast informatica, haalde ik ook een propedeuse Wiskunde. In december 1995 studeerde ik met lof af bij Computer Graphics op het onderwerp "Efficient Evaluation of DMS-splines" (zie [FVW00]). In januari 1996 begon ik als Assistent In Opleiding (AIO) voor het SamenwerkingsOrgaan Brabantse Universiteiten (SOBU) aan project 95AF: "Een ontwikkelomgeving voor correct programma-ontwerp". Van juli 1999 tot juli 2000 werkte ik halftijd als toegevoegd docent aan de TUE (Inf) en halftijd als AIO. Sinds juli 2000 ben ik werkzaam als Universitair Docent (UD) aan de TUE (capaciteitsgroep Informatica).

Cocktails
Visit Cocktail's Homepage
http://www.win.tue.nl/~michaelf/cocktail.html
95-AF