The MacLWB
&
the Logic of Likelihood

Peter Balsiger

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

Leiter der Arbeit:

Prof. Dr. G. Jäger
IAM, Universität Bern

The MacLWB
&
the Logic of Likelihood

Peter Balsiger

von Bern

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

Leiter der Arbeit:

Prof. Dr. G. Jäger
IAM, Universität Bern

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 1. Februar 2001

Der Dekan
Prof. Dr. P. Bochsler

Contents

List of Figures

List of Tables

List of Definitions

List of Theorems

List of Lemmas

List of Corollaries

List of Examples

List of Classes

List of Methods

Introduction

Abstract

This thesis consists of two separate parts. The first part contains information about the MacLWB and its implementation. The creation of the MacLWB required much new implementations, but also forced changes in existing code of the LWB. Furthermore, the first part also gives some general information concerning porting and maintenance, using the LWB as an example.

The second part is completely different, only connected to the first through the common tool used, the Logics Workbench. This second part provides information about a special propositional logic and its implementation in the LWB. The logic of likelihood LL is a modal logic that allows to express statements denoting probability. The second part contains some theoretical aspects of the logic, like the sequent calculus used for the implementation, as well as the practical aspects of the implementation of the prover using object oriented techniques.

The Logics Workbench

The Logics Workbench, short LWB, is an interactive system allowing symbolic computations in various propositional logics. The LWB was first introduced in [26], but has been extended many times.[1] A more detailed overview of the LWB and its internal structure can be found in [43].

The LWB offers the possibility to work in a user-friendly way in classical and non-classical propositional logics, including nonmonotonic approaches. Provided are functions concerning provability, simplification, computation of normal forms, embeddings, and many more. There are functions available for various logics, including classical propositional logic, intuitionistic propositional logic, various modal logics, like K, KT, S_4, and temporal logics. The LWB also provides a programming language, to allow easy extension of the builtin functions.

The LWB was created by a lot of people (cf. `http://www.lwb.unibe.ch/about/authors.html`). The first version was done by Alain Heuerding (cf. [26] and Stefan Schwendimann (cf. [43]). Many years of effort of these and other people were put in the creation of

[1] this thesis uses a beta version of the LWB 1.2

the LWB. Currently, the LWB comprises about 2’500 files with roughly 400’000 lines of code, documentation and tests.

Acknowledgments

A lot of people helped me during the time I worked on this thesis and on the LWB in general. I confine myself to the most important people, because otherwise I’d surely forget someone.

My foremost thanks go to Gerhard Jäeger for many ideas, inspiration and support and also for making the Logics Workbench possible in the first place. Many thanks also to Thomas Strahm, for many interesting discussions and for reading this thesis, Luca Alberucci for a lot of feedback concerning the second part of this thesis, Alain Heuerding and Stefan Schwendimann for creating the first version of the Logics Workbench, Thomas Studer for technical and logical support and the penguins, Wolfgang Heinle for some insights into modal logic , Yvonne Dauwalder for the moral support and Stefan Zapf for proof reading.

Part I

Porting the LWB

Chapter 1

Introduction

This part contains information about the the porting of the graphical user interface of the LWB from a Unix system to the Apple Macintosh. While doing this, some problems surfaced that needed special treatment. These problems and their solutions are presented here. Furthermore, a general overview and description of the implementation of the GUI for the Logics Workbench LWB is given. The material presented here also contains some guidelines and strategies for porting code from one machine to another. Furthermore, information is given describing how to write code that is portable between several systems. Thus, information about the Unix version of the LWB does also appear in this chapter, mainly because of porting to Linux, but the information is limited to those parts affected by porting and maintenance.

1.1 Prerequisites

For the development of the graphical user interface of the MacLWB, certain prerequisites were given. They had to be taken into account when designing the graphical user interface and thus somewhat limited the possibilities for its development.

1.1.1 Existing X WINDOWS GUI

When the development of the graphical user interface for the Macintosh was started, a fully functional, graphical user interface for Unix with X WINDOWS already existed. Therefore, it was obvious to require that the new user interface should at least be similar to the X WINDOWS version. This allows users to switch from one system to the other, without having to learn a new way of using the program. Furthermore, that way some of the documentation and maybe even of the implementation can be shared.

The drawback is that the look and feel of the user interface is already fixed and can only be changed in minor details. Large scale changes would require much change in the existing user interface and thus they are hardly possible. This includes changes that would simplify the implementation on another system, but also prohibits the use of some system specific features.

1.1.2 Existing Code and Algorithms

For the graphical interface, only the appearance was fixed beforehand, the code could be created from scratch. Most of the rest of the code of the LWB had to be reused on the Macintosh. This includes all the logic specific algorithms like provability algorithms, but also most of the kernel and the parser of the LWB, documentation and tests.

Most of these things are not system dependent at all, and thus do not produce porting problems. Unfortunately, there are some small but significant parts of the existing code that turned out to be highly system dependent. These parts had to be identified and adapted to a version compatible with both systems.

1.1.3 Native Look and Feel

The implementation of the MacLWB should create a real, native Macintosh application. Thus, the user interface has to have the same look and feel as any other macintosh application. Because the Macintosh look and feel differs from the one of X WINDOWS, it is not possible to strictly comply with both systems and at the same time produce a user interface that looks and works similar. Therefore, a compromise had to be found between the existing user interface for X WINDOWS and the one for the Macintosh, requiring changes on both sides.

Furthermore, to make sure the application seamlessly integrates in the operating system, some Macintosh specific oddities have to be taken care of as well. This includes program installation and initialization, for example.

Chapter 2

Operating System Comparison

This chapter compares some operating systems, mainly Unix and Macintosh, to show their differences, especially concerning the implementation or porting of applications. The information detailed below is mostly from a developers point of view and not from the one of a user. Furthermore, only the parts somehow concerning the Logics Workbench[1] will be treated.

When porting an application, or even when just developing an application which is intended or expected to be ported, most of the time the following differences are crucial to be known and to be taken care of. Otherwise, at the latest when actually porting an application to another operating system, problems almost certainly arise. Chapter 5 will will actually explain problems when porting programs and how to solve them. This chapter just shows some of the differences between operating systems and how they affect program development.

2.1 Introduction

2.1.1 Unix

There are a lot of different Unix systems available for many different hardware platforms. But many of the concepts and strategies used in all these different systems are the same. The later chapters will only contain general information about Unix, to be able to not have to specify a particular Unix system. Nevertheless, the information presented below is mostly taken from Solaris and Linux systems and may be different for other Unix systems.

[1] the LWB does not support networking, thus this part, for example, is left out.

2.1.2 Macintosh

Contrary to Unix, there is just a single Macintosh operating system, although various versions of that operating system exist. Although there are basically two different hardware platforms that are important for the Macintosh, we will only treat one here, the PowerPC architecture. The older architecture, using Motorola 68’000 processors was, due to compiler problems, never supported by the LWB. Furthermore, most Macintosh computers in use today use the PowerPC architecture anyway.

2.2 Versions

Because operating systems change over time, there are normally various versions of an operating system in use. This may cause serious problems when developing a program for a specific version and then running it on another version, be it newer or older. This has serious impact on how long an application can be used without special maintenance to ensure compatibility.

2.2.1 Unix

The version of the Unix system is not really important when developing for Unix.[2] More important is the version of the different libraries used by a program. Because these libraries are normally not binary compatible between different versions, it is not possible to create a binary release of a program for different versions of a library. For that reason, a lot of Unix programs are distributed in source format and are compiled on the destination machine with the current set of libraries.

2.2.2 Macintosh

On the macintosh the situation is much better, because programs here rarely use external libraries and the operating system is normally upward compatible. Thus, as long as the operating system is not older than the one a program was written for, there should not be any problems. Of course, problems will arise as well if the operating system is much newer.

The first Macintosh operating system, version 0.0 was introduced, along with the first Macintosh in 1984. Since then, several new version were released. Currently, version 9.04 is available. The port of the LWB was done on a MacOS 7.6 system. Some care has been taken to ensure that

[2] actually the version number of a Unix system cannot easily determined, because the version numbers of individual tools greatly differ; the version number can only be used to compare it to versions of the same software producer; comparisons with other Unix implementations of other producers are mostly useless

the Logics Workbench will also run with future versions of the MacOS.[3] It has be successfully tested on a MacOS 8.1 system.

2.3 System Software Routines

The Operating System provides routines to perform basic low-level tasks. This includes, among others, low-level file input and output, memory management, or process and device control. In a way, the operating system provides the communication between an application and the hardware used.

2.3.1 Unix Libraries

The system routines provided by Unix systems are packed together in libraries. These libraries actually are just a bunch of system routines put together in a file. A program using such a library can either statically link the libraries to the executable program, thus actually copying the used system functions into the own program code. Or, as a more modern method, dynamically link the program to the libraries. When the program is run later, the operating system locates the libraries used by a program and dynamically loads the needed functions for use by the program.

Static linking has the big advantage that a program does not need external files to be installed on a computer and thus can be installed more easily. The disadvantage is that libraries used in more than one program have to be stored each time they are used. Thus, a program uses more space on disk and in memory that way.[4]

Dynamic linking removes this disadvantage by using a single library by multiple programs, as well on disk as in memory. The drawback is that the system library must be present when the program is run, or a run time error will occur. Thus, the installation of the program has to make sure that the correct libraries in the correct versions are installed on the system. Furthermore, after a system upgrade it is possible that certain programs stop functioning because libraries are no more present or are not compatible to the old ones.

Unix systems are generally quite strict when dealing with dynamic libraries. When a library is used, the version number of the library has to completely match the one used when developing a program. Otherwise, the library will not be found and the program terminates. This does not make it easy to provide binary releases of a program. Therefore, as mentioned, a lot of programs are distributed as source installations.

[3] the CARBON DATER provided by Apple indicated no serious problems.

[4] this can be quite important, considering that some libraries occupy several MBytes of memory.

Process Manager	launching, scheduling, and termination of applications; information about open processes
Memory Manager	dynamic allocation and release of memory
Virtual Memory Manager	virtual memory services
File Manager	creation, opening, reading, writing, and closing files
Alias Manager	location of specified files, directories, or volumes
Disk Initialization Manager	initialize disks
Device Manager	input from and output to hardware devices
SCSI Manager	information exchange to SCSI devices
Time Manager	periodical execution of routines, execution of routines after a specified delay
Vertical Retrace Manager	synchronize routines with the redrawing of the screen
Shutdown Manager	execution of routines at startup or shutdown

Table 2.1: Some Managers of the Macintosh Operating System

2.3.2 Macintosh Managers

Distinct collections of system software routines on the Macintosh are called managers and get their own names. There is a collection of routines dealing with the initialization of disks, called the *Disk Initialization Manager*.[5] Table 2.1 lists the main managers that are part of the Macintosh operating system. Several Managers are sometimes further grouped into bigger parts, like the TOOLBOX or the FINDER.

For most tasks, higher level routines provided by the Macintosh TOOLBOX (see 2.4) or other services are easier to use and more directly provide the required functionality than the low level operating system functions. While Unix systems distinguish between static and dynamic libraries, managers are always present in the system and may directly be used by any program.[6]

Most of the time, Macintosh managers are compatible between different versions of the operating system. Newer operating system versions may contain new managers or existing managers may get additional functions, but normally old functions are still present for compatibility. Thus, a program developed for a specific version of the MacOS normally runs on newer systems as well. It may have problems running on older systems, though. This makes development of programs running on different version easier, but on the other hand makes the managers quite rigid and with a lot of old routines that are only present for compatibility. Thus, overall, these managers are slower to change than Unix libraries.

[5] in the past, such collections were called packages, a term that is still used for some collections, like—for example—the Standard File Package.

[6] modern MacOS versions do support dynamic and static libraries as well, but these are only used for user defined or third party libraries and not for system functions.

2.3.3 C/C++ Library

The compilers provide several standard libraries supporting the C and C++ standards. While the functionality of these libraries is the same for all Unix systems and for the Macintosh, the implementations are normally not compatible. Thus the program needs to use the compiler dependent libraries to run correctly. This is normally automatically handled by the compiler.

In the case of Unix systems, these libraries can be dynamically linked to the program, resulting in reduced size of the executable file. The drawback is that the user of the program needs to have the same set of C/C++ libraries installed, i.e. an installation of roughly the same version of the compiler. Most compilers do not allow to ship the dynamic libraries with the compiled program, thus the libraries have to linked statically in any case. Fortunately, most compilers allow the dynamic and static linkage of libraries in the same program. If a program is distributed as source, then this is not a problem, because with the recompilation of the program the correct libraries are used automatically. A binary distribution runs into serious trouble, though.

The same is true for modern Macintosh operating systems, where dynamic libraries are available as well. As long as static linking is used, there is no problem, but dynamic linking is even worse than on Unix systems, because contrary to most Unix systems, Macintosh systems don't normally have a compiler installed.

2.4 Windowing System

The windowing system is the part of an operating system most often seen when working with a computer. While the windowing system of different operating systems fulfill about the same tasks, their implementation and most importantly their use from a developers point of view is strongly different.

2.4.1 X WINDOWS

On Unix, the windowing systems is actually divided into two major parts. One part normally is X WINDOWS, while the other part is the overlaying window manager. While there are various different window managers available, the underlying X WINDOWS normally is the same for all Unix systems.[7]

X WINDOWS is standardized and thus can be used from every program without having to worry about a specific type of Unix. On the negative side, X WINDOWS is a fairly old standard and as that does not use modern programming practices. That means that the library is not object oriented and thus is quite hard to use. Furthermore, the available standard libraries do not support sophisticated user interfaces, e.g. dynamic positioning of interface elements. On the other hand,

[7] X WINDOWS is also available for non Unix machines, for example for Windows or MacOS.

X WINDOWS provides a fairly broad standard, which includes network support and is compatible between a variety of Unix systems.

To provide a more comfortable development environment and at the same time give the user a more intuitive and easier to use user interface, there are several window managers available. These are additions to the core X WINDOWS libraries, providing additional functionality. While the basics of all the window managers is always X WINDOWS, the mangers themselves are not compatible to each other. Thus, in order to be able to use a special application, the end user needs to use the same window manager.[8]

A variety of window managers with associated interface libraries are available. Below is a short description of two major products using two different approaches.

MOTIF

MOTIF is a quite old extension to X WINDOWS and allows much easier handling of a graphical user interface and is also available for most Unix systems. Furthermore, it is possible to link it statically to the final program, removing the requirement to have MOTIF installed.

On the other hand, MOTIF has some of the same drawbacks of X WINDOWS, because it does not use object oriented techniques. Furthermore, MOTIF is not freely distributable and thus can only be redistributed using static linking. The graphical version of the LWB was implemented using the MOTIF interface library.

QT

QT itself is only an interface library, but several window managers were built using this library, most notably KDE for Linux. QT is completely object oriented and supports dynamic placement and runtime creation of interface elements. It provides an abstract graphical user interface layer, to implement the final user interface. This makes the porting of a program to another operating system supporting QT easy and simplifies the integration of the user interface into an object oriented C++ program. QT itself is available for all Unix systems as well as for various Windows systems. Furthermore, at least the Unix version of the Qt development libraries are available for free.

2.4.2 FINDER

The FINDER is the most visible part of the MacOS. It is responsible for managing the user's desktop display. The FINDER uses many of the other parts of the operating system, like the

[8] it is possible to create an application with static linking which runs on any X WINDOW system, but this application might have a look and feel totally different to other applications installed on the system because of incompatible window managers.

QuickDraw	screen display operations (drawing of graphics or text)
Window Manager	creation and management of all kinds of windows
Dialog Manager	creation and management of dialog boxes
Control Manager	creation and management of controls (buttons, checkboxes, radio buttons, pup-up menus, scroll bars)
Menu Manager	creation of the menu bar, handling of drawing and actions within menus
Event Manager	reporting of events, communication with other applications
TextEdit	simple text-formatting and text-editing (input, selection, cutting, pasting)
Resource Manager	reading and writing of resources
Finder Interface	interaction with the FINDER
Scrap Manager	cutting and pasting among applications
Standard File Package	standard dialog boxes for file selection
Help Manager	Balloon Help
List Manager	creating of lists of items
Sound Manager	sound output
Sound Input Manager	sound input

Table 2.2: Macintosh TOOLBOX Managers

Macintosh TOOLBOX or QUICKDRAW, to keep track of all files stored on a Macintosh. The FINDER also takes care of copying and moving files, or creating folders.

As mentioned, the FINDER uses other components to provide its functions. The two most important of these parts are described below.

TOOLBOX

The TOOLBOX is a set of routines provided for developers to create user interfaces. The TOOLBOX is not visible as an individual part of the MacOS or its interface, because unlike the FINDER it is not present on the disk and there is no single program using it. Instead, it is used to create the user interface of most applications, thus it visible in most applications. Actually, it is the TOOLBOX that ensures the Macintosh look and feel, i.e. the TOOLBOX makes sure that all Macintosh applications look and work similar. It mainly deals with the management of the user interface, i.e. helps to establish the connection between the application and the user.

The Macintosh TOOLBOX contains a variety of Managers. Table 2.2 lists the most commonly used TOOLBOX Managers. It is a level above the Operating System, thus it uses some of the low-level functions provided by the Operating System to provide its functionality.

QUICKDRAW

Where the TOOLBOX provides high level support for graphical user interface elements, QUICKDRAW provides the routines to do basic drawing on the screen. Thus, it is comparable to X WINDOWS on Unix Systems, although much of the functionality present in X WINDOWS is distributed into several distinct parts on the Macintosh.

2.5 Files

The handling and internal structure of files and their names varies greatly from operating system to operating system. Unfortunately, even modern programming languages don't support a uniform handling of files and their names.

Actually, there are two different situations when developing a program where file names are important. First, when a program is able to use files, it needs a way to determine the name of the file to read or write. Second, the source code of the program most certainly needs to include text from external files, which must again be referenced from the code itself. We will deal with this second problem later, while looking at the different developing environments for the operating systems (see chapter 3).

2.5.1 Unix Files

A Unix file is a simple block of data, without additional, file specific information. Unix can only distinguish different file types by directly looking at the contents of the file (i.e. magic numbers). While this is bad for applications dealing with files, it is a great benefit when copying files between different operating systems. Because no additional information is present, the files can be copied as is.

There are still some considerations to be taken care of when dealing with Unix files on different operating systems. When dealing with ASCII files, the ending of a line is unfortunately not standardized. While Unix uses a line feed to represent a new line, the Macintosh uses a carriage return and Windows even uses a carriage return and a line feed. Thus copying an ASCII file from one operating system to another might make problems.[9]

Even more problematic are binary files, i.e. files where data is not stored in text format but instead as it is present in the memory used by the program. In such cases, the processor used determines how integer values are stored. Different processor, even using the same operating system, may interpret the values differently. That's why most Unix programs use the text format to store data instead of directly using a binary format.

[9] programs to transfer files, like `ftp` automatically convert line endings in ASCII files depending on the systems involved

2.5.2 Mac Files

The file concept of the Macintosh operating system differs in some key concepts from other operating systems. The most important difference is the concept of resources. While a file on most other operating systems, including Unix and Windows, consists of a single part, a Macintosh file basically consists of two separate parts. One part contains the normal contents of the file, like on other operating systems. The second part of a file, contains additional information concerning the file. This information is stored in the so called resource fork and contains a variable number of resources. This resource fork makes it impossible to copy a Macintosh file to another operating system without conversion. At least the resource fork has to be stripped from the file to get the real data to another operating system. In that process, some vital information of the file may be lost. Thus, even simple text files cannot be easily converted from Macintosh to Unix.

Macintosh files have another specialty. While Unix and Windows use a file's extension or some magic number at the beginning of a file to determine its type and which application to use to open the file, the Macintosh has a creator and a file type stored in a resource. Both are 4 Byte character sequences, uniquely identifying the program that created the file and the file's type. With this information it's easy to determine how to treat a file. Using this information, it is possible for the MacLWB to start the LWB when a file is double clicked. In such a case, the LWB is started and the selected files are automatically executed. With this mechanism, it is possible to implement a sort of scripting for the LWB, i.e. automatically calling a predefined set of commands at startup.

2.5.3 File and Path Names

Of course, the names of the files also differ from one operating system to another. On Unix, directories are delimited with the slash '/', while DOS/Windows uses the backslash '\' and the Macintosh uses a colon ':'. Furthermore, the maximal length allowed for a file name differs from operating system to operating system as well.[10]

2.6 Software Installation

The installation of a program is an important aspect of a program's distribution. Several key questions have to be answered before a program can be released. Most importantly, if the program is to be released as a source or a binary distribution. The latter has the advantage of much easier installation, while the earlier is more compatible with different versions of an operating system (cf. 2.2 and 2.3).

The installation of a program is highly system dependent. As already mentioned, a program is normally written for a specific version of an operating system and using it on a newer, or worse

[10] fortunately, the very restricting 8/3 limits of DOS are finally gone on Windows systems; thus file names up to a length of 32 characters don't make any problems.

on an older version of the same operating system might or might not work. Below we have a look at the two different basic distribution methods and how feasible they are for Unix or Macintosh systems.

2.6.1 Source Distributions

A source distribution of a program contains the complete source code and all the information necessary to build the final executable program. The end user needs the appropriate compiler and all other additional tools necessary to built the executable program from the sources.[11]

There are several tools that help to make building the final executable program easier. Nevertheless, the building process can be complicated, time consuming and error-prone. On the other hand, source distributions have many advantages that often make them the only choice for a program's distribution. The main advantage surely is the compatibility to the current operating system. When a program is compiled on the machine it is intended to run on, a lot of compatibility problems are solved. The version problems with libraries on Unix systems (cf. 2.2) are solved, thus prohibiting compatibility problems with system software routines and problems with positions of dynamic libraries on disk.

Of course, the source distribution has to be prepared for the desired target operating systems or compilation almost certainly won't work. The building process on a specific Unix system may work if it was prepared for another Unix system. But, actually compiling a program for a system it was not originally intended for generally requires a port of the program which involves a great deal of work.

A natural requirement when installing a program from a source distribution is the presence of an appropriate development environment. While such an environment can be assumed to be present on most Unix systems, most Macintosh or Windows systems don't have a development environment installed and thus are not a good choice for source distributions.

2.6.2 Binary Distribution

The binary distribution only contains the final executable program and all the files necessary to run the program. The source code of the program is not part of a binary distribution.

Normally, a binary distribution only contains an executable program for a specific operating system and a specific hardware platform. It is generally not possible to run the same program on a different hardware platform or on a different operating system. This, of course, is the main drawback of binary distributions. Their advantage, on the other side, is generally a quite easy installation. With binary distributions it is, with some effort, possible to create installation packages that are easy to install and don't require special skills or efforts when installing.

[11] like the MAKE tool

Because there are a lot of different computer systems and most of them are incompatible, a program may have to offer many different binary distributions in order to be able to allow it to be run on a variety of computer systems. This requires huge maintenance effort and thus is rarely done for Unix systems. For Macintosh systems, on the other hand, this kind of software distribution normally is the only possible way.

2.6.3 Comparison

For the Macintosh, only binary distributions make sense. It cannot be assumed that the user of a program has access to a C++ development environment. Furthermore, Macintosh systems tend to be quite compatible between different operating system versions (compare 2.2.2). Thus, a binary distribution is much easier for the user and does not have many drawbacks.

This is different for Unix systems. The various Unix systems are not binary compatible between each other, mainly because the processors used are not compatible. Furthermore, Unix systems tend to be quite strict concerning library versions (cf. 2.2.1). Thus a lot of programs are distributed as source code, which the user has to compile to be able to use the program. This has the advantage of providing a single installation file which will then, after compilation, run on any Unix machine. The developer has to make sure, that the compilation runs without problems on any system, which can be challenging task.

Nevertheless, the LWB is distributed using binary packages. Mainly because distribution as source would make the build process much more difficult and would require drastic changes. See chapter 3 for more information about the tools used for creating distribution packages.

2.7 Scripting and Redirection

Scripting describes the process of automatically control one or more programs with a predefined sequence of commands. With scripting it is possible to call programs and process their results with other programs. Furthermore, it is possible to generate a sequence of commands which can then be run without user intervention to carry through a time consuming or repeating event.

Redirection is used to change where input to a program comes from or where output goes to. With redirection, a program can take its input from a file or another program instead from user interaction. Similarly, output can be stored in a file or redirected to another program instead of displaying it on screen.

For the LWB, scripting is mainly used to automatically test the internal algorithms for correctness. This is done by issuing commands to the LWB and compare the results to a previously computed and hand-checked result. For LWB users, scripting and redirection allows the automatic execution of LWB commands stored in a file, for example to carry through time consuming computations without requiring direct user interaction.

2.7.1 Unix Shell

The shell is an important part of a Unix system. It provides a text oriented way of invoking and controlling programs.

Input and output using a shell are standardized and an integral part of the C++ language. Thus, an application using only shell in- and output can easily be ported to another computer system supporting C++ and shells. Furthermore, using a shell makes scripting of programs and redirection of input and output easy possible, without additional implementations.

While these programs are often not as user friendly as programs with a graphical user interface, they can still be quite easy to use. Furthermore, using the shell has several advantages from the developers viewpoint.

By using scripting and input/output redirecting it is quite easy to implement procedures that automatically test a program for errors. That way, after changes are made, it can easily and automatically be checked if all that worked before still does.

The drawback of shell oriented programs is of course the lack of a comfortable user interface. Thus it is normally best, as was done with the LWB, to implement two version of a program, one using Shell input and output (the ASCII LWB) and another one using a more sophisticated graphical user interface (the XLWB and the MacLWB).

2.7.2 Apple Script

The Macintosh operating systems does not contain a shell. Because the user interface of the Macintosh is completely graphic oriented, a shell would be out of place. Nevertheless, there is still a way to use scripting on the Macintosh, although a more complicated one. The Macintosh supports Apple Script, a Macintosh specific scripting system. While it does not allow to redirect input or output, it can be used to control the execution of a program.

The drawback of this approach is that only programs actually supporting Apple Script can be controlled and only those features of a program can be used which are supported via a command in the Apple Scripting Language. Furthermore, the this language is quite complicated and uses a complicated method of writing script files.

All in all, scripting on the Macintosh is much harder to use than it is on a Unix system. Because the LWB mostly uses scripting for automatic testing, it is not supported on the Macintosh. The automatic tests can be carried through on Unix much easier and because the program code corresponding to the LWB algorithms is the same for Unix and the Macintosh, the tests actually test the Macintosh version of the LWB as well. The automatic executing of certain commands, is solved on the Macintosh by the so called initialization files, i.e. files that are automatically read with the MacLWB when double clicked.

2.8 Compilation and Building

We look at two principally different approaches for the building process of a program. The first is shell oriented and the second graphic oriented.

2.8.1 Shell Oriented

The shell oriented approach uses makefiles, the MAKE tool and the shell to create an application (see chapter 3). The makefile contains the commands and rules necessary to create an application from its source files. MAKE is then able to create the application, using the compiler and all other necessary tools. It has the capability of automatically deciding which files need to be remade because their sources were changed. This makes the tool very powerful and can drastically reduce the time required for compilation.

The MAKE tool cannot only be used for the creation of the program itself. It can also generate distribution archives, run test procedures, create documentation files and a lot of other tasks that can be started in a shell. The creation of correctly working makefiles for the building process can be quite complicated and time consuming. On the other hand, all tasks required for compilation and building of a program and its maintenance can be done using MAKE. This can greatly decrease maintenance overhead and reduce the number for errors.

All in all, MAKE is a powerful tool, capable of integrating many other programs. It is available for all Unix systems and used for almost all programs.

2.8.2 Graphic Oriented

Because some operating systems—like the Macintosh—do not provide a shell, another than shell oriented approach has to be used. A graphic oriented development environment includes handling of projects and their building and compiling process using a completely graphic oriented user interface. The developer only has to select the source files to be compiled into the final program. Building and linking is done automatically by the development environment.

Clearly, this type of building is much easier, because most of things necessary are done by the development environment and do not concern the developer. There are several drawbacks, though. Only tasks that are actually supported by the development environment can be done. This is normally only compiling and linking of a single program. If there are tasks that need to be done in order to create a final program that are not supported by the development environment, they have to be done using external programs and, worse, have to be repeated by hand whenever necessary. Such things are, for example, the automatic generation of source code, like a parser generated with LEX (cf. 3.2.5) and YACC (cf. 3.2.6), or the automatic generation of program documentation. While such things can be integrated into shell oriented development environments, there is normally no way to include them in a graphic oriented development environment.

Furthermore, graphic oriented development environment use proprietary data structures and files to store project specific information. Thus, the building information is not compatible with other development environments. If a program is to be developed for different computer systems, then either the same development environment is available for all systems, or multiple systems have to be maintained.

2.9 Preferences and Configuration

The configuration values, i.e. preferences, of a program are stored at different position and in different ways on Unix and on Macintosh systems. The Macintosh uses the preferences folder, while Unix generally uses dot files.

2.9.1 Preferences Folder

The preferences folder is a special directory on the hard disk of a Macintosh system. This folder is used to store the preferences information of all programs installed on the system. This way, the program as well as the user know where to look for program specific configuration files.

The data is stored in standard Macintosh files using resources. The Macintosh operating system provides only some very basic functions to deal with preferences. This is mainly a way of locating the preferences folder, because its name and location are user defined. Handling of the preferences values is left to the program, with no additional support by the operating system.

2.9.2 Dot Files

The standard for configuration files on Unix systems are so called dot files. These are files beginning with a dot '.'. They are normally not shown when displaying the contents of a directory and are thus hidden from the user.

The files normally contain ASCII text defining the configuration values of a program. Normally, a program reads a specific dot file at startup. That way, it is possible to set the configuration of a program permanently. The location of the dot files is not as standardized as Macintosh preferences files, but the files are generally stored either in the users home directory, a program specific configuration directory or in the program directory itself.

2.10 Documentation

Because operating systems are quite different from each other, there also exists different documentations for the systems. Below are some of the standard documentations essential for Unix and Macintosh software development.

2.10.1 C++

Most general C/C++ books assume Unix systems for development and thus provide enough information to start programming on such systems. Furthermore, the input/output system of C++[12] was actually made for Unix and thus is easy to use on Unix systems. On the Macintosh, the books can be used for general C/C++ programming, but additional information is necessary for input and output, because standard input and output may not be available.

To be able to use operating specific routines and libraries, additional books are required, even if coding in C++. Thus books for X WINDOWS or MOTIF are necessary for Unix systems and documentation for general Macintosh programming is almost certainly required to develop Macintosh Software.

2.10.2 System Software Routines

The documentation for calling system software routines is usually either available online on the Internet or on the system itself. This information normally is enough to be able to call a specific system function, but additional documentation may be necessary to be able to really use these functions effectively.

Man Pages

Unix uses the man pages for system documentation. The man pages allow easy and fast access to the most important information concerning a system routine. Additional information must be found either on the Internet or in books about Unix development.

Inside Macintosh

Inside Macintosh is a library of books forming a complete reference manual of the system software of the Macintosh. The books are primarily designed as reference books, not as step-by-step tutorials. One notable exception to this rule is the introductory book *Inside Macintosh: Overview* [6]. It does not contain references to a specific part of the macintosh operating system, but a general introduction to programming on Macintosh computers and to the other Inside Macintosh books.

The Inside Macintosh library consists of approximately 35 volumes[13], each with several hundred pages describing a specific part of the Macintosh operating system. They contain a reference of all available functions and data structures. There are books for text processing [10], imaging [12], memory [14], files [11] and all the other parts of the operating system. The whole contents

[12] the `iostreams`.

[13] see the bibliography for a complete list of available volumes

of these books is electronically available [5]. Furthermore, the most recent additions to the Inside Macintosh library are available on the word wide web only.

The books are, like the whole operating system, mainly written for Pascal or Assembler programmers. There are always some side references about using the functions and data structures in C, but object oriented techniques or paradigms are neither used nor supported. This makes usage of some of the functions a bit unpredictable at first, for example when dealing with Pascal or C Strings (cf. 2.12). The functions normally just assume one or the other, without clearly stating which.

One of the most important volumes in the Inside Macintosh series, apart from the introductory volume [6], is the volume describing the Macintosh TOOLBOX [13]. It contains the references to the most important routines of the operating system. The contents of the volume is continued in [9]. For application containing text or graphics, the volumes [12] and [10] are important. Most application will also need information from [14], [11], and [16].

2.11 Memory

Every application has to deal with memory in a way. Most of the time memory will handled care of by the compiler. Even if the compiler does not have complete memory management with garbage collection, like C/C++, most memory allocations and its uses are still handled. Only freeing of memory must concern a C++ developer.

Below are some consideration concerning allocation and use of memory for Unix and the Macintosh. The release of memory is not specially treated, because this is a general C++ problem, discussed in any basic book about C++.

2.11.1 Unix

A program running on a Unix system can assume that it has complete control of all memory available in the system. The individual memory spaces used by different programs are all managed by the operating system and need not concern the program or its developer. An application cannot, even if faulty, write into the memory space of another program. Virtual memory is automatically handled by the operating system, actually providing the Unix program with more memory than is available as physical memory of the machine. Additionally, a memory block is never moved by the operating system.

All this makes memory management for Unix programs easy and most of details are handled by the C++ compiler. A C++ program written for Unix thus has to do nothing special to allocate or free memory, even when calling system software routines.

2.11.2 Macintosh

The Macintosh uses a cooperative multitasking (see below) to share the hardware between several processes. Therefore, an application can only use part of the total available memory. Some of the total RAM available on a machine is used by the operating system, while the rest is shared among all open applications.

The whole memory is split into two sections, called partitions, the system partition and the application partitions. The system partition mainly consists of a system heap and a set of global variables. The system heap is reserved for exclusive use by the operating system and other system software components. The set of global variables, called system global variables (or low-memory system global variables), are used to maintain different kinds of information about the operating environment. For example, the `Ticks` global variable contains the number of ticks ($\frac{1}{60}$ of a second) that have elapsed since system startup. Additionally, pointers to the heads of various operating system queues are also stored in system global variables. The system partition is not normally used by applications, except in some rare cases when reading some variables.

All memory outside the system partition is available to applications and other software components. When an application is launched, the operating system assigns it a section of memory known as its application partition, which normally is the only memory usable by an application. Therefore, an application has a fix amount of memory, determined beforehand[14]. The application heap will be allocated at the bottom of the partition, growing upward, while the stack starts at the end of the partition and grows down. The operating system makes sure that the heap does not grow above a predefined limit (the `ApplLimit`), but it does not prevent the stack of growing into the heap, but instead checks approximately 60 times a second if the stack has moved into the heap. A system error is generated if it has. The application stack mainly contains the memory used for the execution of functions, i.e. arguments and return values of functions, as well as local variables. The application heap, on the other side contains the data that is dynamically allocated during process execution, like window records, dialog records or document data.

The system used by the Macintosh makes handling of low memory usage absolutely necessary. Because the memory a program can use usually is limited beforehand, it is always possible that memory runs out. On most Unix and similar systems, the available memory generally is so big that most developers don't check if an allocation of memory actually succeeded. On the Macintosh this kind of carelessness can be fatal.

The Memory Manager does all the necessary bookkeeping about free and used memory blocks. Contrary to the stack, the allocation and freeing of memory can occur in any order. Thus, the heap can, after the application has been running for a while, become fragmented into a patchwork of allocated and free blocks. This fragmentation is known as heap fragmentation. If the memory is fragmented too much, the system may no more be able to satisfy a request for a single, large block of memory, even if the total amount of free memory is much larger than the desired block. If this happens, the Memory Manager tries to collect the free space into a single block,

[14] this amount is fixed in the information panel of a program icon in the FINDER.

an operation which is known as heap compaction. To be able to do this, the blocks of memory used by the application need to be relocatable, i.e. the operating system must be allowed to move memory blocks. This heavily concerns application programs. Data allocated by an application may suddenly be moved by the operating system, making all pointers to this data invalid.[15] To prevent this, the Macintosh operating system supports a special data type, the handles. These are actually double indirected pointers, which are adjusted when the operating system moves a memory block. This prevents these pointer from becoming invalid, but on the other hand requires additional tasks to be done when allocating, freeing or using such memory in a program.

The system used by the Macintosh operating system is quite old and originates from a time where memory was scarce and it was useful to save memory. Most modern computer systems have enough memory nowadays and thus most of these techniques have become obsolete. Furthermore, the virtual memory mechanism used in current computers does similar things on the hardware level. This solution is much faster and furthermore does not burden the developer with additional considerations to take care of.

2.12 Data Types

C++ uses various data types that may be problematic when porting programs from one system to another. Below, we briefly look at some standard data types generally used in programs that might produce problems.

2.12.1 Pointers

Pointers are normally no problem, because all systems know this data type and support it in the same way. Depending on the architecture, pointers may have different sizes though, but this does not normally create any problems if the code is written properly.

Handles

Handles are pointers to pointers, indirectly giving access to memory blocks. Unix programs rarely use handles to access memory. They are mainly used on the Macintosh, because of the relocatable memory blocks. Because handles are just pointers to pointers, they can be used on any system, but rarely make sense unless required by the operating system.

[15] in C++ this means all data not stored on the stack

2.12.2 Strings

C Type

Unix uses C type strings, i.e. strings which are terminated by a NULL character. This is the same used by C++ and thus there are no problems using strings on a Unix system, even when calling system software routines.

The object oriented string class provided by C++ cannot directly be used to call system functions. But it provides a means of converting the string into a standard C string, which can be safely used with any system function.

Pascal Type

Although some of the new the operating system routines of the Macintosh use C type strings as well, most still use use Pascal type strings. These strings contain the length of the string in the first byte of the text, but are not NULL terminated.

While the text itself is completely the same for both types of strings, the values can nevertheless not be interchanged. Thus, special care has to be taken when using a string obtained by a Macintosh system function in a C string function and vice versa. Normally, the string has to be converted, or output may not be as desired.

Additionally, because the length of a string is stored in the first byte, a Pascal type string may not be longer than 255 characters. For longer strings, a different data structure has to be used.

2.12.3 Integers

Integers are a bit more complicated, because they more closely depend on the processor used. Mainly the size of an integer may vary from one system to another. That can be a problem, if a computation produces numbers that are bigger than those that can be stored on a specific processor.

Another problem with integers is the internal ordering of the value in memory. This ordering differs from processor to processor. Normally all memory access is handled by the compiler, but in some special cases problems might arise. This is for example the case, if a memory block is directly stored in a file. While this file can be read without any problems on a similar processor, the values might be interpreted completely different on another machine.

These problems are the same for Unix and Macintosh machines. Normally, they are not to hard to solve. If no data is interchanged between systems in binary format, the latter problem disappears. If code is not written using the full storage capabilities of integer values up to the limits, but instead in a way that only uses the standard 4 Byte integers, then the size considerations are not a real problem as well.

Of course, similar problems arise for floating point numbers, but they can be solved accordingly.

2.12.4 Complex Types

More complex types are normally built up using the basic types and thus produce no additional problems. There may be some problems if compilers handle some parts of the language differently, which rarely occurs. The problem most often encountered is when a compiler does not support one of the more recent additions to the C++ standard. Using such features results in a program that cannot be compiled on all compilers.

The internal layout of complex data types is highly compiler dependent, though. Thus, it is never a good idea to store the values of a complex data type in binary format by simply storing its memory. This most certainly results in unportable programs. Furthermore, it is never guaranteed that a future version of the same compiler will actually place the data in completely the same order. Thus, binary files written like that might become unreadable when the program is compiled with a new version of the compiler.

2.13 Interprocess Communication

Interprocess communication is used every time a program or process tries to communicate with another one. There are several solution how this communication can be carried through. The most important ones will be detailed below.

For the LWB, interprocess communication is important in several parts. The same type of communication is used to get information from the operating system to the graphical user interface. Furthermore, interprocess communication is used when a program needs to be interrupted by the user. Lastly, interprocess communication is used to communicate with external programs, like the web browser used for the help system.

2.13.1 Signals

A Unix system uses signals to provide simple communication to a program. This only allows very basic, one way communication. All that can be sent to another process is a signal, a simple number. The system provides various predefined signals, for example to interrupt, quit, abort, kill, alarm, restart or stop a process.

The signal is received by a process, which should react appropriately. Some of the basic standard signals, like stopping, are automatically handled by the C library, thus need not be handled by the program code.

2.13.2 Events

The Macintosh does not use signals for interprocess communication but instead uses a similar, though more versatile event system. Events are complex data types used to exchange information between processes. An application can receive or send many types of events. They are usually divided into three categories, low-level events, operating system events, and high-level events.

Low-level events are created by the operating system for simple, hardware or user generated information. Low-level events are sent by the Event Manager when the user presses a mouse button, releases the mouse button, presses a key on the keyboard, or inserts a disk. Furthermore, the Event Manager sends an application an event when the window is to be activated or when part of a window has to be redrawn. If an application requests an event and there is none, a null event is returned by the operating system.

The Event Manager sends an operating system event when an application's processing status is about to change or has changed. For example, an operating system event is sent to an application that is brought into the foreground by the user. The application then has to reactivate itself.

High-level events are sent to an application by another application or another process. They are mainly used for interprocess communication on a higher level.

The communication between two processes (or a process and the operating system) conforms to the client/server model. The communication is normally initiated by a client process sending a request to a server application using an Apple event. Both processes can run on the same computer or on remote computers connected over a network.

[39] contains a standard vocabulary of Apple events that can be used to for the communication between applications. These are predefined events, corresponding to many services an application might request of another. Additional events can be defined by an application, but communication is only possible if both processes interpret the event the same way..

Event Loop

The central part of most Macintosh applications is the event loop. This is the loop in the program in which all events are treated. The event loop initiates the appropriate reaction to all events received by a program. This approach is especially useful for applications with a graphical user interface, because all user actions are transmitted to the program via events. The event loop will treat these events and thus makes the program able to react to user input.

The process of obtaining and reacting to events is active, the program needs to poll events in order to treat them. As long as no events are read, no user input reaches the application. Because of the Macintosh's cooperative multitasking (see next section), the computer looks frozen when an application seizes the CPU for an extended time. To prevent this, each time events are requested from the event manager, the CPU switches to the operating system, giving it some time to update the display and do other important things. Thus, the event loop also helps to keep the user interface alive. This only works, if each reaction to an event does not use more than some fraction of a second of CPU time. Otherwise, the display may freeze.[16]

2.13.3 Shell and I/O Redirection

There are many situations, where an application needs to be able to start an external program to carry through some specific task. Because this is an important situation, C/C++ has a special, system dependent function defined just for this case. The `system()` function starts an external program, independent from the current one. Technically, this function takes a string and treats it just as if it was entered into a shell (cf. 2.7.1).

Using this method, it is easy to start external programs and with I/O redirection it is even possible to get back some output of the program after the external program is finished.

The LWB needs to communicate with a web browser for the help system and with the ProofWish tool to display classical proofs. On the Macintosh, this is handled using the events described above. On Unix, signals cannot be used to achieve the desired result. Thus, using `system()`, the shell is used.

2.14 Processes

Most modern operating systems support multi tasking, i.e. allow multiple programs to be executed simultaneously. Even if just a single program is running on the computer, normally other, system specific processes are running as well. The operating system needs a means to switch between these process, because most machines only have a single processor and thus can execute a single program at a time.

2.14.1 Cooperative Multitasking

The Macintosh supports a simple form of multi tasking, so called cooperative multi tasking. As the name implies, cooperative multi tasking requires the cooperating of all tasks running on a machine. Thus, each application is required to hand on the CPU to other tasks from time to time, to give each other task a possibility to do something. The operating system relies on each individual task to be cooperative, i.e. an application can never be forced to give up the CPU but instead should do it on its own.

Cooperative multi tasking puts several requirements on an application, essential for smooth operation of the whole machine. The Macintosh operating system adds further requirements to the application to give the user a better handling of multiple programs running at the same time.

[16] this is contrary to the Unix signals, which can interrupt program execution and force a program into a certain state.

These requirements will be detailed below, mainly because ignoring some of these requirements results in serious troubles for the operating system.

Application States

The Macintosh distinguishes two main states for an application. An application can either be the active application, a foreground or a background application.

The active application is the application that currently interacts with the user. It is not necessarily the application that currently has control over the processor, i.e. the application currently running. It is gets all events generated because of user actions, like mouse or keyboard events. If a process[17] is currently in control of the CPU, it is called the foreground process. As mentioned above, the active application may also be the foreground process, but this is not necessary. A process that is open and ready to run but isn't the foreground process, is called a background process.

Context Switches

As long as an application runs, it can only be interrupted by hardware interrupts. To give time to background processes, the application needs to call the Event Manager from time to time. That way, the application gives away the control of the CPU for a short period, during which another process gets some time for computations. The process of changing the CPU from a currently running process to a process waiting for execution is called a context switch.

There are two distinguishable types of context switches, major and minor switches. A major switch fully switches from one application to another, maybe for a long period. For that switch, the application has to move it's windows from the back to the front or vice versa. Furthermore, some windows may be hidden or shown again. Thus the active application switches from one process to another.

For a minor switch, the Process Manager gives a background process some computing time, without actually changing the active application. The windows are not changed in any way for such a switch and the background application is still not capable of getting user input.

Events

Several special events are used to change an applications processing status. If an application should be switched into the background, the Process Manager sends it a suspend event. This tells the application to prepare to give the CPU away. The switch is carried though the next time the application checks for an event. If an application is switched into the foreground, it gets a resume event, *after* the switch.

[17] an application or a desk accessory.

If an application gets a suspend event, it should deactivate its front window, and remove the highlighting from any selections and hide all floating windows. After a resume event, the application should activate its front window and restore any states inside the windows.

One special type of event is the null event. If an application receives a null event, this means that no other application needs the CPU for computation. Thus the application can then perform idle processing. This includes blinking the insertion point, for example. The application should do only minimal processing at this time though, because some other process might want to compute something soon. If an application is in the background, it can do other processing while in the background. It should not perform lengthy tasks, because these might slow down responsiveness of the foreground process. Furthermore, an application should never interact with the user if it is in the background.

Summary

All in all, the Macintosh's process handling system is quite old fashioned and puts a lot of requirements to the programs running on the system. Thus, if an application is faulty, locks and hang-ups often occur and cannot be prevented. Furthermore, there is no real process and resource management. A process may not easily be killed and there is no way of giving execution priorities to processes. Last but not least, there is no real interrupt system, making it hard to control a program which runs out of bounds (the only solution on the Macintosh is to completely terminate the program).

2.14.2 Preemptive Multitasking

An operating system supporting preemptive multi tasking is capable of switching from one process to another, without having to rely on the cooperating of the processes involved. Thus, the operating system switches from one process to another using its own scheduling information. It is capable of interrupting a running application and switch to another one.[18]

Thus, a program can be written without regard to other programs or the operating system. The program is assumed to have complete control of the whole machine. The process switching is done by the operating system without the program even noticing it.[19] Furthermore, a program cannot normally prohibit process switching, and thus even if a program crashes, the machine might go on. More importantly, development is easier because no care has to be taken to consider process switching. On the other hand, this solution requires a complicated operating systems and puts special requirements to the hardware.

All Unix systems support preemptive multi tasking.

[18] this is normally done through hardware interrupts.

[19] the program can notice it if need be, but normally this is not necessary.

2.15 Compatibility

It is quite important for an application to be compatible with future versions of the operating system. As already mentioned (cf. 2.2), this task is quite hard on Unix systems but feasible on Macintosh systems.

2.15.1 Macintosh

To ensure compatibility for a Macintosh application means to write applications that are able to run with little or no modifications on all members of the Macintosh computer family and on all system software versions. There are some basic guidelines to be followed to ensure compatibility:

- never directly address the hardware
- never directly write to the screen
- don't rely on system global variables

By keeping these guidelines in mind, an application will most certainly run with future Macintosh systems. Furthermore, Apple released a special program, called `Carbon Dater`, which checks an application program for its fitness with new versions of the operating system, especially System X.

2.15.2 Unix

Unix applications are not normally binary compatible with major changes in the Unix system. While small changes or updates normally don't interfere with programs, major changes won't allow the program to go on working, mainly because the necessary libraries are no more available or are not compatible with old ones.

The solution to this problem is the recompilation of the program. This normally solves all compatibility programs, because now the current, updated system libraries are used. Because the C++ standard and the compilers normally change very slowly, compilation is normally no problem on a new system.

Chapter 3

Development Tools

This chapter gives a brief overview of the tools used for the creation of the MacLWB and the Unix LWB. For further, more detailed information, have a look at the program documentations.

3.1 Macintosh

3.1.1 CodeWarrior

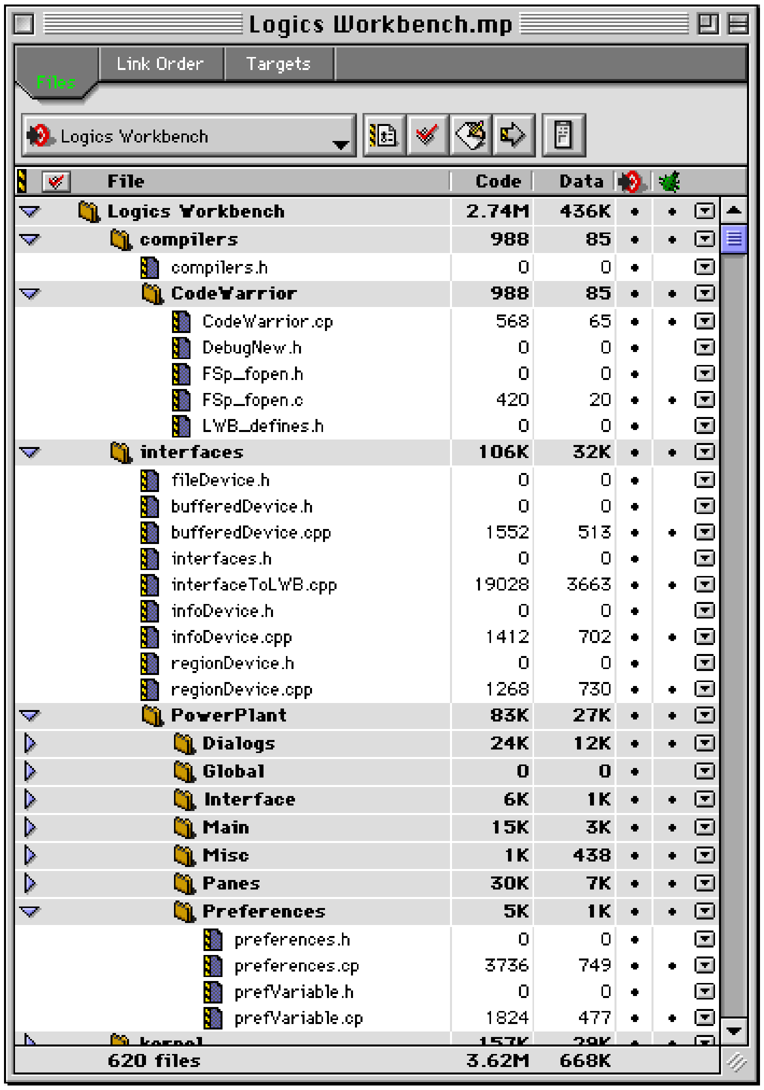
CodeWarrior by Metrowerks is a commercial integrated development environment for the Macintosh. It has a completely graphical user interface and includes an editor for writing the source code, a compiler to create the executable files and a debugger, in case the program does not work as expected. It also includes automatic project management. This means, the compiler will automatically decide which files need to be recompiled and does automatically link the appropriate files to create an executable program.

The integrated development environment does not allow additional steps to be taken into the computation process. Thus it is not possible to create source files when building an application, like using `lex` (cf. 3.2.5) or `yacc` (cf. 3.2.6) to automatically create a parser.[1]

The integrated compiler is a complete C/C++ compiler supporting all current standards of C++, including function templates and the standard template library (STL). Furthermore, it has special additions dealing with Macintosh specific issues, like handling Pascal type strings or resources.

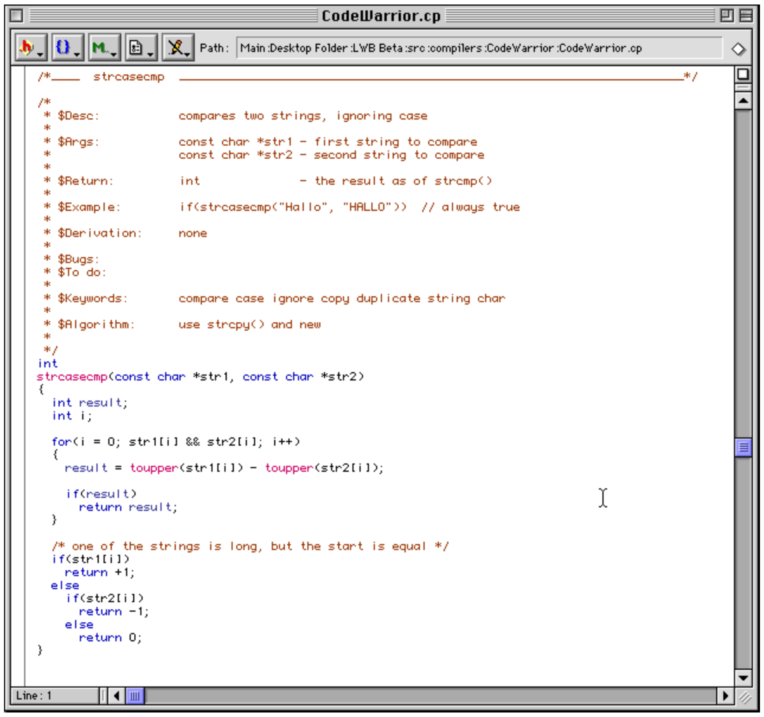
Picture 3.1 shows the project management of CodeWarrior, picture 3.2 displays a sample of an editing session, and picture 3.3 shows the debugger when debugging the MacLWB. CodeWarrior contains extensive online documentation but further books like [23] are also available.

[1] these source files had to reused over from Unix.



File	Code	Data		
Logics Workbench	2.74M	436K	•	•
compilers	988	85	•	•
compilers.h	0	0	•	
CodeWarrior	988	85	•	•
CodeWarrior.cp	568	65	•	•
DebugNew.h	0	0	•	
FSp_fopen.h	0	0	•	
FSp_fopen.c	420	20	•	•
LWB_defines.h	0	0	•	
interfaces	106K	32K	•	•
fileDevice.h	0	0	•	
bufferedDevice.h	0	0	•	
bufferedDevice.cpp	1552	513	•	•
interfaces.h	0	0	•	
interfaceToLWB.cpp	19028	3663	•	•
infoDevice.h	0	0	•	
infoDevice.cpp	1412	702	•	•
regionDevice.h	0	0	•	
regionDevice.cpp	1268	730	•	•
PowerPlant	83K	27K	•	•
Dialogs	24K	12K	•	•
Global	0	0	•	
Interface	6K	1K	•	•
Main	15K	3K	•	•
Misc	1K	438	•	•
Panes	30K	7K	•	•
Preferences	5K	1K	•	•
preferences.h	0	0	•	
preferences.cp	3736	749	•	•
prefVariable.h	0	0	•	
prefVariable.cp	1824	477	•	•
kernel	157K	29K	•	•
620 files	3.62M	668K		

Figure 3.1: CODEWARRIOR: graphical project management


CodeWarrior.cp
Path: Main:Desktop Folder:LWB Beta:src:compilers:CodeWarrior:CodeWarrior.cp
Line: 1

```
/*____  strcasecmp  ______________________________________________________*/

/*
 * $Desc:          compares two strings, ignoring case
 *
 * $Args:          const char *str1 - first string to compare
 *                 const char *str2 - second string to compare
 *
 * $Return:        int              - the result as of strcmp()
 *
 * $Example:       if(strcasecmp("Hallo", "HALLO"))  // always true
 *
 * $Derivation:    none
 *
 * $Bugs:
 * $To do:
 *
 * $Keywords:      compare case ignore copy duplicate string char
 *
 * $Algorithm:     use strcpy() and new
 *
 */
int
strcasecmp(const char *str1, const char *str2)
{
  int result;
  int i;

  for(i = 0; str1[i] && str2[i]; i++)
  {
    result = toupper(str1[i]) - toupper(str2[i]);

    if(result)
      return result;
  }

  /* one of the strings is long, but the start is equal */
  if(str1[i])
    return +1;
  else
    if(str2[i])
      return -1;
    else
      return 0;
}
```

Figure 3.2: CODEWARRIOR: source code editing

3.1.2 POWERPLANT

POWERPLANT is part of the CODEWARRIOR development environment. It is an extensive library for the creation of graphical user interfaces (cf. [24]).

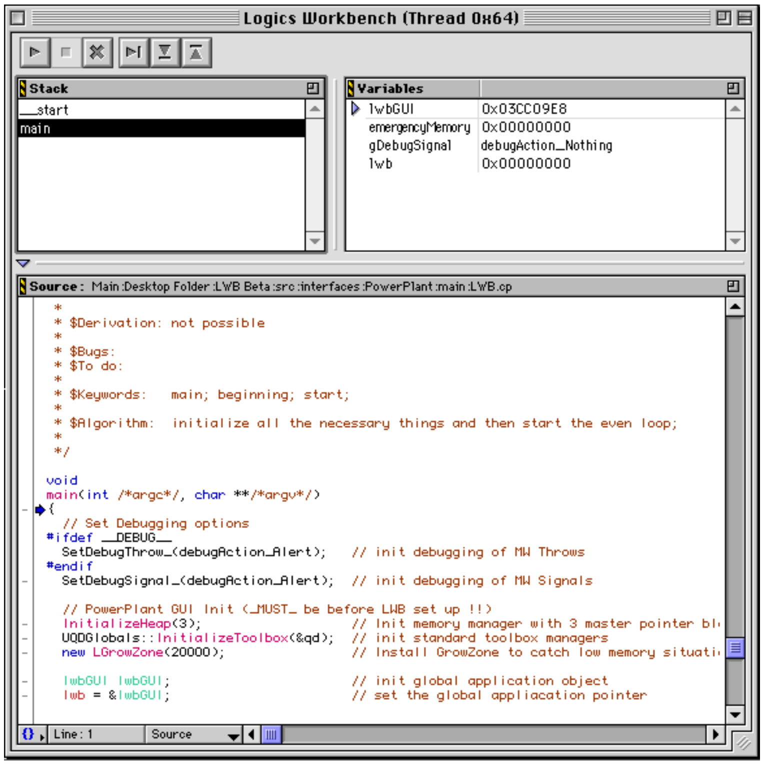
Overview

Although the Macintosh Toolbox already provides quite a lot of routines and support to create graphical user interfaces, it does not directly support object oriented programming and thus is complicated to use. POWERPLANT smoothes these problems by providing a layer between a graphical user interface and the Macintosh operating system. This layer is completely object oriented and thus can be easily introduced into any C++ program.

Furthermore, the CONSTRUCTOR tool (see 3.1.3) allows to graphically create a user interface using the POWERPLANT libraries.

Structure

The POWERPLANT library is written in C++ and its source code is available to developers, making debugging and problem solving easier. The library bases all graphical user interface


Logics Workbench (Thread 0x64)
Stack
Variables
Source : Main :Desktop Folder :LWB Beta :src :interfaces :PowerPlant :main :LWB.cp
Line : 1
Source

```
__start
main
```

```
lwbGUI           0x03CC09E8
emergencyMemory  0x00000000
gDebugSignal     debugAction_Nothing
lwb              0x00000000
```

```
 *
 * $Derivation: not possible
 *
 * $Bugs:
 * $To do:
 *
 * $Keywords:   main; beginning; start;
 *
 * $Algorithm:  initialize all the necessary things and then start the even loop;
 *
 */

void
main(int /*argc*/, char **/*argv*/)
{
  // Set Debugging options
#ifdef __DEBUG__
  SetDebugThrow_(debugAction_Alert);   // init debugging of MW Throws
#endif
  SetDebugSignal_(debugAction_Alert);  // init debugging of MW Signals

  // PowerPlant GUI Init (_MUST_ be before LWB set up !!)
  InitializeHeap(3);                   // Init memory manager with 3 master pointer bl
  UQDGlobals::InitializeToolbox(&qd);  // init standard toolbox managers
  new LGrowZone(20000);                // Install GrowZone to catch low memory situati

  lwbGUI lwbGUI;                       // init global application object
  lwb = &lwbGUI;                       // set the global appliacation pointer
```

Figure 3.3: CODEWARRIOR: a debugging session

elements on a pane, defined by the class `LPane`. A pane is a simple display element, including a size and position and the methods necessary to paint, move and otherwise handle it. All other graphical elements are derived from that base object. For example, the object for views (`LView`) is directly derived from a pane (`LPane`) and additionally provides a means of displaying a pane that is larger than its display and whose visible portion can be scrolled.

If a graphical interface element is required that is not directly supported by POWERPLANT or if one of the provided elements needs to be changed somewhat, the existing classes can be derived to create new, user defined custom classes. These classes can then be integrated into the POWERPLANT library structure just like the internal classes. Even the graphical construction tool, the CONSTRUCTOR, can support custom defined classes in a limited way.

The library has some predefined structures which must be taken into account when developing applications, to ensure that own classes are smoothly integrated into the POWERPLANT environment.[2] This requires the implementation of some necessary methods in custom interface classes.

Problems

Unfortunately, The POWERPLANT library has some drawbacks as well. One of its most serious drawbacks is the big overhead used for drawing interface elements. POWERPLANT uses quite inefficient ways for drawing and often draws a single pane several times. Together with the already inefficient interface handling of the Macintosh operating system and a slow computer, this results in painfully slow display updates. Especially when dealing with a big number of interface elements, as is the case with the MacLWB and its input and output regions, this can become unbearable.

Furthermore, POWERPLANT does not completely support all available Macintosh managers. Thus it can be necessary to directly deal with a manager. While this itself may be inconvenient, it gets really problematic if directly calling some manager routines interferes with POWERPLANT's internal data structures. Unfortunately, this is often the case and can most of the time can only be solved by reimplementing some parts of POWERPLANT.

A major drawback and especially time consuming were some errors in the library, which resulted in faulty handling of interface elements. While most of these errors were minor, they still caused quite a lot of debugging. Most of these minor errors were corrected in newer versions of the POWERPLANT library, directly leading into new problems. New versions of the POWERPLANT library were quite incompatible to previous versions and the changes sometimes were quite severe. This required the rewriting of some parts of the graphical user interface with each change of library version.

One last drawback is that the library is not portable, i.e. there is only a version for the Macintosh. If the same library was also available for Unix or Windows system, then porting a program—including the graphical user interface—would be easy and not time consuming. Furthermore,

[2] Examples are the calling chain used to process keyboard and menu commands or the drawing procedure used to draw the individual interface elements on the screen.

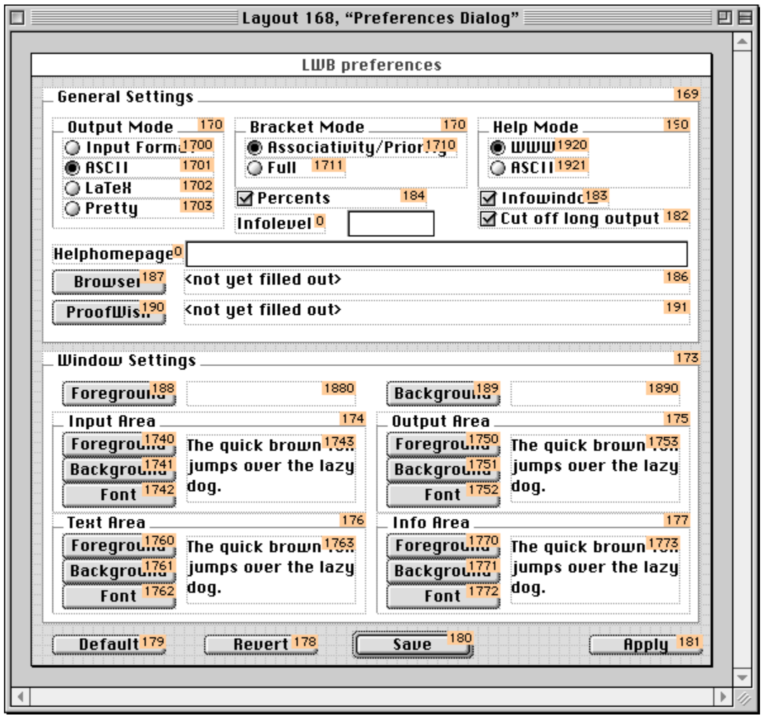


Figure 3.4: CONSTRUCTOR: graphical interface creation

changes in the user interface would not only benefit a single system but all systems where the interface library was used.

3.1.3 CONSTRUCTOR

The CONSTRUCTOR is a development tool closely working together with the POWERPLANT library. It is a graphic oriented tool (see figure 3.4) for the creation of graphical user interfaces using POWERPLANT. The interface is composed by putting interface elements of various predefined types at their places and define all their values. From a thus designed user interface, the CONSTRUCTOR creates the necessary resources, which can be added to an application program. From these resources, the POWERPLANT library automatically constructs all objects necessary to display the graphical user interface on the screen.

The CONSTRUCTOR even allows the handling of custom classes. This is done when providing the appropriate information to the CONSTRUCTOR by creating a new type of interface element. The program gets the base class of the new interface element to define its general behavior and additionally all newly added variables of the class. CONSTRUCTOR then allows the positioning of these new custom elements just as the predefined elements, additionally with the new variables.[3]

[3] Other, special properties of the new elements are not shown in the created interface, though. It is only possible to

```
// a type for a window dimension
type TYPE_DIMENSION
{
  unsigned integer;         // width
  unsigned integer;         // height
  unsigned longint;         // top position
  unsigned longint;         // left position
};

resource TYPE_DIMENSION (PREF_DIM_MAIN, "Position of Main Window")
{
  DEF_DIM_MAIN_WIDTH,    DEF_DIM_MAIN_HEIGHT,
  DEF_DIM_MAIN_TOP,      DEF_DIM_MAIN_LEFT
};

resource TYPE_DIMENSION (PREF_DIM_INFO, "Position of Info Window")
{
  DEF_DIM_INFO_WIDTH,    DEF_DIM_INFO_HEIGHT,
  DEF_DIM_INFO_TOP,      DEF_DIM_INFO_LEFT
};
```

Table 3.1: REZ: Definition of the Dimension type and some resources

The CONSTRUCTOR has one major drawback, sharing it with the POWERPLANT library. It does not allow the relative positioning of interface elements. Thus, only static user interfaces can be built using the CONSTRUCTOR. If an interface element has to be positioned relative to another one, especially one that may change its size, or if the size or position of an interface element has to react to changes in the layout of the user interface, then the positioning and moving of the element has to be done in the program code an cannot be left to POWERPLANT or the CONSTRUCTOR.

3.1.4 REZ

The REZ tool provides a resource description language that can be used to create Macintosh resources. It uses a C preprocessor like language to generate the final resources. REZ also allows the definition of user defined types and resources of such custom types. The REZ compiler takes a description file and generates a file containing the defined resources.

Defining a resource with REZ may be a bit more complicated than doing the same with a tool using a graphical user interface[4], but using the REZ language has the advantage of automation. Resources can be generated automatically, without user interaction. Thus, certain elements of the resources may depend on other parts of the program, including values from a definition file. This

define new variables.

[4] like CONSTRUCTOR or RESEDIT

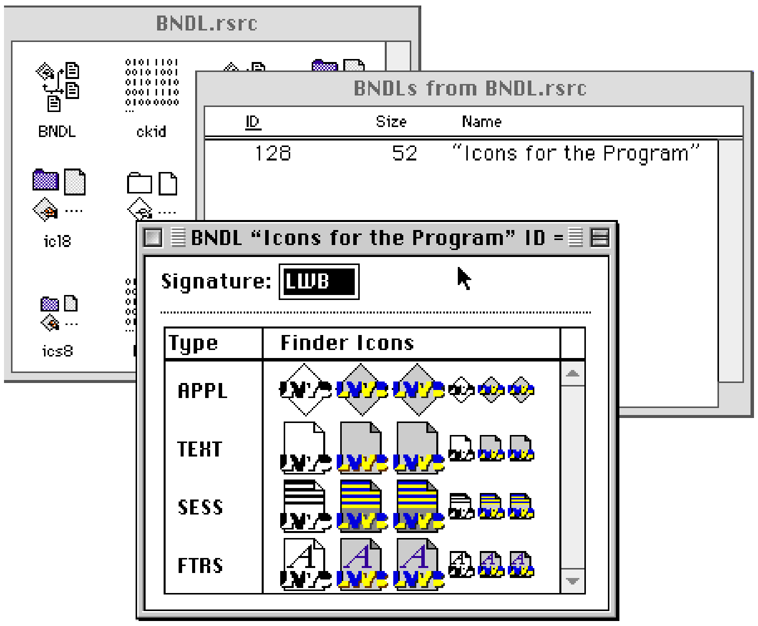


Figure 3.5: RESEDIT: graphical resource editing

way, information has to be stored and maintained at a single place in the source code, making maintenance much easier. Furthermore, when changing the information, automatically all similar information is changed throughout the whole program.

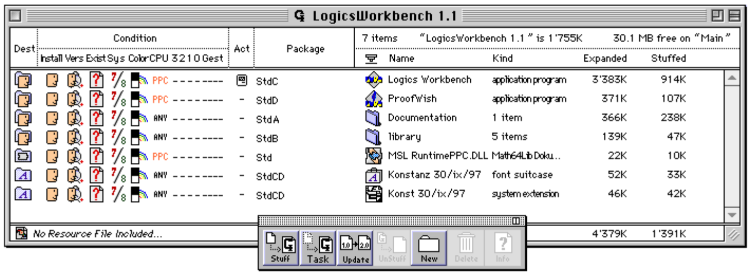
3.1.5 RESEDIT

Like REZ, RESEDIT is used to create Macintosh resources. Unlike REZ, RESEDIT is a graphical resource editor. It shows all the resources of a file and allows changing, deleting, and adding of resource values. This prevents automatic generation of resources, but on the other hand allows easier definition, especially for icons and other pictures. Like REZ, RESEDIT does also allow the definition and treatment of custom resources.

Luckily, CODEWARRIOR allows multiple resource files to be included in a single executable. Thus it is possible to generate part of the resources using Rez or the CONSTRUCTOR and other resources using RESEDIT.

3.1.6 INSTALLER MAKER

The INSTALLER MAKER is used to generate an installable archive of a program. This archive is distributed to the end user for installation on the target machine.


LogicsWorkbench 1.1

7 items "LogicsWorkbench 1.1" is 1'755K 30.1 MB free on "Main"

Condition CPU	Package	Name	Kind	Expanded	Stuffed
PPC --------	StdC	Logics Workbench	application program	3'383K	914K
PPC --------	StdD	ProofWish	application program	371K	107K
ANY --------	StdA	Documentation	1 item	366K	238K
ANY --------	StdB	library	5 items	139K	47K
PPC --------	Std	MSL RuntimePPC.DLL	Math64Lib Doku...	22K	10K
ANY --------	StdCD	Konstanz 30/ix/97	font suitcase	52K	33K
ANY --------	StdCD	Konst 30/ix/97	system extension	46K	42K

No Resource File Included... 4'379K 1'391K

Figure 3.6: INSTALLERMAKER: installation package creation

The generation of the installation package is easily done using a graphical user interface. The files to be installed can be placed in the archive, along with additional information, for example where the file will be installed. This allows files to be installed in predefined locations, like fonts in the fonts folder or preferences in the preferences folder.

The final installation package makes the installation of a program simple. All a user has to do is to double click on the archive icon, select an installation directory where the application should be installed (or accepting the default location) and all the rest is done automatically.

Unfortunately, the INSTALLER MAKER does not allow the automized creation of installation packages. Thus, for each new release or update of a program, the package has to be newly assembled by hand. This makes the process of creating installation packages error prone and time consuming.

3.1.7 TCL/TK

TCL/TK is a scripting language with user interface support, available for a variety of operating systems. The language allows the creation of programs with a user interface that may be run on any implementation of TCL/TK on any of the supported systems.[5]

This type of user interface seems to be a clever way to create portable programs that may be run on different systems. It has several drawbacks, though. First, the language is a scripting language and thus quite slow. Furthermore, it is not fully object oriented, making the development of larger programs hardly possible.

But even using TCL/TK only for some small tasks is not without problems. While the language is theoretically system independent, several limits still exist. Not all that is possible on a Unix system can be done on a Macintosh and vice versa. Thus, an application has to be developed with all target systems in mind and must be thoroughly tested on each system to make sure that the program will smoothly run on all of them.

[5] Unix, Macintosh, Windows

3.2 Unix

3.2.1 SUN VISUAL WORKSHOP

SUN VISUAL WORKSHOP is a commercial C and C++ development environment by SUN MICROSYSTEMS. It contains an integrated development environment and the CC compiler. The compiler in its current version 5.0 does support most of the current C++ standard.[6] The compiler is quite slow in compiling code with templates and sometimes uses a lot of memory. Furthermore, program compiled with CC that use the standard template library (STL) are quite slow, especially when compared to other compilers, like GNU GCC.

On the positive side, the VISUAL WORKSHOP comes with a lot of additional tools convenient for development. For example, it contains an integrated development environment, a debugger and performance and benchmarking tools.

3.2.2 GNU GCC/EGCS

Contrary to CC, the GNU GCC[7] is freely available. It can be used on any Unix system and on a lot of other systems, like Windows, as well.

The implementation of the standard template library within GCC is quite fast, especially when compared to CC. The compilation of the source code is fast as well. On the negative side, compiling with optimization turned on is very slow and requires up huge amounts of memory.[8]

GCC does not come with a set of tools and programs to help in the development process. Instead, the compiler relies on the many freely available tools that can be used for programming. Sometimes, these tools lack the professional quality of a commercial product, but because there are a lot of them, the right tools can almost always be found.

3.2.3 GNU MAKE

As all other GNU tools, GNU MAKE[9] is freely distributed using the GNU license. The program is available for all Unix systems.

The program is used to automate program building and compilation. It provides a language to define rules and commands for building files from other files. When run, it does automatically

[6] earlier versions of the compiler lacked some important parts of the standard, like template functions and the standard template library (STL).

[7] previously called EGCS

[8] more than 256 MBytes at least

[9] SUN CC also contains a MAKE tool, which supports much less features and is much less sophisticated.

```
.
.
.

GLOBAL_DIR      = /home/lwb/beta/src#   # dir of global lwb
LOCAL_DIR       = /home/lwb/beta/src#   # dir of local installation
TMP_DIR         = /home/til/betatmp#    # dir for temporary files
X_DIR           = /usr/openwin#         # X directory

.
.
.

%.cpp: $(RCS_DIR)/%.cpp$(RCS_EXT)
        @$(CO) $< $@                    # generate C++ file from RCS
%.h: $(RCS_DIR)/%.h$(RCS_EXT)
        @$(CO) $< $@                    # generate header from RCS
$(DEST_DIR)/%.o : %.cpp
        @$(PRINT) "$(INDENT)  compiling $<"
        @$(CXX) -c $(CXX_FLAGS) $(CXX_LIBFLAGS) $(CUR_INC) -o $@ $<
                                        # C++ files
.
.
.
```

Table 3.2: GNU MAKE: excerpts of an example makefile

compare file modification dates to determine which files need to be remade and which files are up to date.

The features supported by GNU MAKE are many, thus writing complex makefiles is possible. It is possible to integrate almost any other, shell based tool into the building process. Thus, it is easy to integrate RCS, LEX, YACC and the various compilers into the building process of a program. MAKE does automatically generate source files with these programs and compiles them into a final executable.

MAKE is not limited to building of programs, though. Any task that depends on files to be created by shell tools using some predefined rules can be done by MAKE. Thus, cleaning the directory tree, testing a program, creating distribution packages, making backups, layouting documentation and a lot of other things can all be automated using MAKE.

Figure 3.2 shows some small excerpts from the standard makefile used to for the building process of the Logics Workbench.

3.2.4 EMACS

Because the GCC does not have an integrated development environment, a separate editor has to be used. EMACS is the standard and very powerful editor for Unix. It is available for free for Unix and also for Windows. It contains an own programming language and a lot of predefined features helping in various editing tasks[10], including program development.

3.2.5 LEX

LEX is a freely available tool for the automatic generation of a lexical analyzer. Together with YACC it can be used to automatically generate a parser from a given set of rules. LEX creates fast and powerful lexical analyzers, but only produces C code and is not object oriented. This makes integrating the lexical analyzer into an object oriented program complicated.

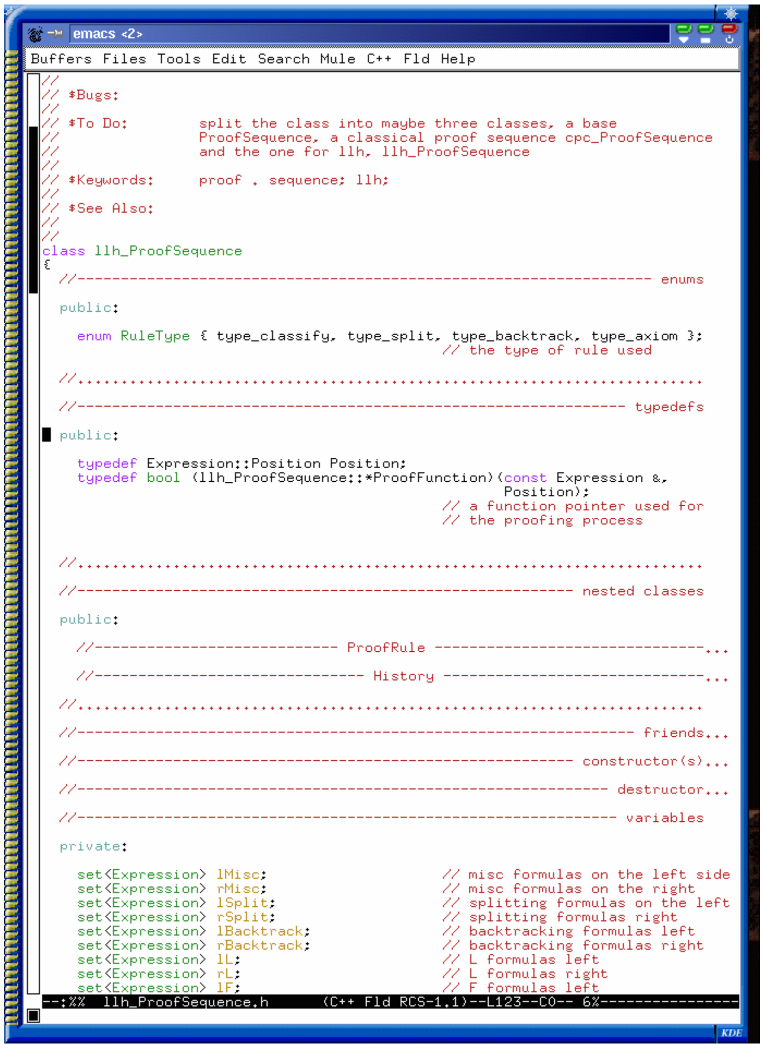
Table 3.3 contains some excerpts from the file containing the rules for the lexical analyzer of the Logics Workbench.

3.2.6 YACC

As a lot of Unix tools, YACC is available for free. YACC takes a file containing rules describing a language and produces C source for a parser capable of interpreting text for this language.

YACC can use LEX to lexically analyze the text before its is actually parsed. As LEX, it does not produce object oriented code and is thus problematic to integrate into an object oriented program.

[10] like text coloring or automatic indentation.



```
//
// $Bugs:
//
// $To Do:        split the class into maybe three classes, a base
//                ProofSequence, a classical proof sequence cpc_ProofSequence
//                and the one for llh, llh_ProofSequence
//
// $Keywords:     proof . sequence; llh;
//
// $See Also:
//
//
class llh_ProofSequence
{
  //------------------------------------------------------------------ enums

  public:

    enum RuleType { type_classify, type_split, type_backtrack, type_axiom };
                                          // the type of rule used

  //......................................................................

  //--------------------------------------------------------------- typedefs

  public:

    typedef Expression::Position Position;
    typedef bool (llh_ProofSequence::*ProofFunction)(const Expression &,
                                               Position);
                                          // a function pointer used for
                                          // the proofing process


  //......................................................................

  //--------------------------------------------------------- nested classes

  public:

    //--------------------------- ProofRule -------------------------------...

    //------------------------------ History ------------------------------...

  //......................................................................

  //------------------------------------------------------------------ friends...

  //--------------------------------------------------------- constructor(s)...

  //------------------------------------------------------------- destructor...

  //-------------------------------------------------------------- variables

  private:

    set<Expression> lMisc;                // misc formulas on the left side
    set<Expression> rMisc;                // misc formulas on the right
    set<Expression> lSplit;               // splitting formulas on the left
    set<Expression> rSplit;               // splitting formulas right
    set<Expression> lBacktrack;           // backtracking formulas left
    set<Expression> rBacktrack;           // backtracking formulas right
    set<Expression> lL;                   // L formulas left
    set<Expression> rL;                   // L formulas right
    set<Expression> lF;                   // F formulas left
--:%%  llh_ProofSequence.h      (C++ Fld RCS-1.1)--L123--C0-- 6%----------------
```

Figure 3.7: EMACS: source code editing

```
.
.
.

alpha       [a-zA-Z]
digit       [0-9]
integer     {digit}*
symbol      {alpha}({alpha}|{digit}|_)*
.
.
.

"="             { tokenpos += yyleng; return tEQUAL; }
"<>"            { tokenpos += yyleng; return tNONEQUAL; }
">="            { tokenpos += yyleng; return tGREATEREQ; }
">"             { tokenpos += yyleng; return tGREATER; }
"<"             { tokenpos += yyleng; return tLESS; }
"<="            { tokenpos += yyleng; return tLESSEQ; }

{help}          { tokenpos += yyleng; return tHELP; }
quit|bye|g2h    { tokenpos += yyleng; quitReadFlag = true; return 0; }

plus|"+"        { tokenpos += yyleng; return tPLUS;  }
"-"             { tokenpos += yyleng; return tMINUS;  }
mult            { tokenpos += yyleng; return tMULT;   }
div             { tokenpos += yyleng; return tDIV;    }
mod             { tokenpos += yyleng; return tMOD;    }

.
.
.
```

Table 3.3: LEX: excerpts of an example lex rules file

```
.
.
.

%token tPROC tBEGIN tEND tLOCAL tVAR tRETURN tIF tTHEN tELSE
%token tFOREACH tIN tDO tRAISEERROR tCATCHERROR tWHILE tAPPEND
%token tPUSH tPOP tRANGE tEVAL tBY tTO tFOR tARRAY tINC tDEC
.
.
.

arg_decl_list :
  tSYMBOL              { if(declare_argument_value($1, 1)) YYERROR;
                           free($1); $$ = 1; }
  | tVAR tSYMBOL       { if(declare_argument_reference($2, 1)) YYERROR;
                           free($2); $$ = 1; }
  | arg_decl_list ',' tSYMBOL { if(declare_argument_value($3,$1+1))
                                  YYERROR;
                                free($3); $$ = $1 + 1; }
  | arg_decl_list ',' tVAR tSYMBOL {
                            if(declare_argument_reference($4,$1+1))
                              YYERROR;
                            free($4);
                            $$ = $1 + 1; }
;

opt_arg_decl_list :
  tSYMBOL                         { if (declare_opt_argument($1))
                                      YYERROR;
                                    free($1); $$ = 1; }
  | opt_arg_decl_list ',' tSYMBOL { if (declare_opt_argument($3))
                                      YYERROR;
                                    free($3); $$ = $1 + 1; }
;
.
.
.
```

Table 3.4: YACC: excerpts of an example yacc rules file

Table 3.3 contains some excerpts from the file containing the rules for the parser of the Logics Workbench.

3.2.7 MOTIFATION

MOTIFATION is a commercial tool used to create graphical user interfaces for Unix machines with X WINDOWS and MOTIF. The user interface is created with a graphical editor by placing the desired interface elements at their positions. The C code generated by MOTIFATION has then to be completed with own functions.

The C code obtained by MOTIFATION is not object oriented and almost impossible to read. Therefore, it is problematic to integrate such code into an object oriented program.

3.2.8 REDHAT PACKAGE MANAGER (RPM)

The REDHAT PACKAGE MANAGER (RPM) was originally developed for Linux systems, but because it is freely available as open source, has in the meantime been ported to various other Unix systems. It is the standard installation package manager for most Linux systems. From a specification file, the program creates installation packages of programs for their distribution. It is also used to install the package on the target machine. RPM does support additional tasks to be carried out when installing a package as well as the possibility to install different files in different destination directories. It is able to check that the correct libraries or other programs are installed on a system before installing a program.

The program automates the creation of installation packages and makes final installation an easy and controlled task. It also allows the removal of programs after their installation and other administration tasks. Furthermore, it is also possible to distribute source packages with RPM, which can automatically be built on the target machine.

Table 3.5 contains the beginning of the specification file used to create the Linux distribution package of the Logics Workbench.

3.2.9 TAR and GZIP

TAR and GZIP are both small, freely distributable Unix tools. They can each be used alone, but are mostly used together to created compressed archives of multiple files.[11]

TAR is used to pack multiple files together into a single file. Originally, this was used to make backups of files to tapes, but nowadays it is more often used to create archives of files. TAR itself does not compress the resulting file. Thats what GZIP is for. It takes a file and compresses it, generating a compressed version of the file.

[11] modern versions of TAR can automatically call GZIP when creating archives

```
Summary: The Logics Workbench
Name: lwb
Version: 1.1
Release: 1
Copyright: distributable
Group: Applications/Math
Source: ftp.iam.unibe.ch:/pub/LWB/lwb_1.1_source.tar.gz
Icon: lwbicon.gif
URL: http://lwbwww.unibe.ch:8080/
Vendor: University of Bern
Packager: Peter Balsiger <balsiger@iam.unibe.ch>
ExclusiveOS: Linux
Prefix: /usr/local/lwb-1.1
BuildRoot:

%description
The LWB offers the possibility to work in a user-friendly way in
classical and non-classical propositional logics, including nonmonotonic
approaches.

%prep
%setup -c
cd src
co Makefile.linuxlocal
co Makefile.Xlinuxlocal
co Makefile
co Makefile.default

%build
cd src
export RPM_BUILD_DIR
make -r SYS=linux --no-print-directory prepare
make -r SYS=linux --no-print-directory
make -r SYS=Xlinux --no-print-directory prepare
make -r SYS=Xlinux --no-print-directory

%install
if !test -d $RPM_BUILD_ROOT/usr; then mkdir $RPM_BUILD_ROOT/usr; fi
if !test -d $RPM_BUILD_ROOT/usr/local; then
  mkdir $RPM_BUILD_ROOT/usr/local; fi
mkdir $RPM_BUILD_ROOT/usr/local/lwb-1.1
ln -s $RPM_BUILD_ROOT/usr/local/lwb-1.1 $RPM_BUILD_ROOT/usr/local/lwb
mkdir $RPM_BUILD_ROOT/usr/local/lwb-1.1/bin
cp $RPM_BUILD_DIR/lwb-1.1/bin/lwb $RPM_BUILD_ROOT/usr/local/lwb-1.1//bin
.
.
.
```

Table 3.5: RPM: beginning of a specification

Using the TAR-GZIP combination can create simple distribution packages of a program, containing all required files. With TAR-GZIP it is not possible to create sophisticated installations. It is not possible to easily distribute files into different directors or to execute some pre- or post-installation steps. On the other hand, creating and unpacking a TAR-GZIP archive is easy and fast.

3.2.10 RCS

RCS is the abbreviation for REVISION CONTROL SYSTEM. It is another freely distributable Unix tool, used to manage different versions of files, mainly source code of programs.

It allows the space efficient and controlled storage of multiple versions of a file, including the handling of revision numbers and log entries. Furthermore, RCS solves most problems occurring when different people work on the same files. In such a case, it has to be made sure, that not two persons change a file at the same time. RCS solves this by providing a locking mechanism, which only allows a single user to edit a file at a time.

In order to use RCS, all users need access to the same storage area, where the master files for each file are stored. Thus, RCS cannot be used over a network without direct access to a common file server.

3.2.11 CVS

CVS stands for CONCURRENT VERSIONS SYSTEMS and is another free Unix tool. It is a enhancement of RCS and is actually based on it.

The focus of CVS is on distributed development. Thus, it supports a client/server mechanism for file modification. Users connected over a network can all work together on the same sets of files at the same time. CVS does not, like RCS, normally lock files while they are modified, to allow all users to change files when necessary. Instead, CVS contains a powerful merging strategy. Files updated by two users at the same time are automatically merged into a single file containing all changes. If changes overlap, CVS is not able to automatically merge the files and thus inserts special comments to make merging by hand easier. CVS provides additional features, like the support of multiple releases of a program or different development branches.

3.2.12 HSC

HSC is a small, simple HTML preprocessor. It generates HTML files from HSC source files.

HSC understands all current HTML statements and automatically checks that they are correctly used. Furthermore, HSC understands additional statements that are translated into HTML text when the files are generated. These statements allow the definition of macros, including loops

```
<$INCLUDE FILE="macros_titlepage.hsc">

<PAGE TITLE    = "The Logics Workbench"
      DESC     = "the main page of the LWB Documentation"
      KEYWORDS = "logic, propositional, computational, symbolic,
                  education; decision procedures, simplification
                  of formulas, proofs, normal forms, embeddings;
                  classical, non-classical, nonclassical,
                  intuitionistic, modal, provability, multimodal,
                  tense, linear; non-monotonic, nonmonotonic,
                  autoepistemic, circumscription, closed world
                  assumption, default logic">

<TABLE>
<TABLERULE>
<WIDTHLINE VALIGN=MIDDLE><A HREF=":about/index.html">
      <IMG ALIGN=MIDDLE SRC=":pics/information.gif" ALT="about"
           BORDER=0></A>
  <TD><A HREF=":about/index.html">About the LWB</A>
  <TD><A HREF=":new.html"><IMG ALIGN=MIDDLE SRC=":pics/new.gif"
       ALT="News" BORDER=0></A>
  <TD><A HREF=":new.html">What’s New</A>
<TABLERULE>
<WIDTHLINE VALIGN=MIDDLE>
.
.
.
```

Table 3.6: HSC: sources for a HTML page

and conditions. It is even possible to directly include text from external programs right into the HTML text.

HSC is mainly used to generate static HTML pages and to check their correct use of the HTML syntax. It also allows to create a common and easily changeable layout of pages by using macros instead of directly using HTML statements.

Table 3.6 show the beginning of the source of the LWB home page. The '`<Page>`' statement at the beginning is a macro that automatically sets up the general layout of the page.

3.2.13 TCL/TK

TCL/TK as described in the previous chapter for the Macintosh is also available for Unix systems. It is generally the same as the Macintosh version.

Chapter 4

Parts of the LWB

Before we look at the different steps done for porting the LWB, we take a closer look at those parts of the LWB that are most affected by porting. A complete overview of the LWB can be found in [26] and [43].

4.1 Installation

The installation of the LWB is a crucial step for the program development. It has to make sure that the program is correctly installed and run on the target system. Furthermore, the installation has to be as easy as possible, to make it feasible for the end user.

The Macintosh Installation of the LWB is done with the Installer Maker (cf. 3.1.6) and the Unix installation is done either using a compressed archive (for Solaris, cf. 3.2.9) or an `Rpm` package (for Linux, cf. 3.2.8).

The installation process of the LWB has to meet some requirements to ensure correct installation and to allow running the LWB without problems.

4.1.1 Compression

The final installation file needs to be compressed for several reasons. A compressed package needs less memory on the server offering it for download. But more importantly, the download time for the user to get the installation package onto the target computer is much less with a compressed package. Furthermore, the installation package should consist of a single file. Otherwise, the user has to download and take care of too many files. Forgetting one of the installation files would result in serious installation problems.

4.1.2 Installation Directories

The LWB contains some files that have to interact in a special way with the operating system. These are mainly the configuration files, fonts and dynamic libraries. It depends on the operating system where these files have to be put after installation.

On Unix, the files have to be stored either in a directory already containing similar files or the environment variables pointing to these files have to be adjusted.[1] Because the LWB for Unix is also distributed as a simple compressed archive, only the second solution is feasible. The installation itself cannot adjust the environment variables, therefore this has to be done when the LWB is started (see the next section).

On the Macintosh, the situation is different. Because a special installation tool is used to install the application on the target machine, files can be distributed to the appropriate directories. This is actually necessary on the Macintosh, because the MacOS does not have something similar to environment variables. Fonts and preferences files have to be stored in their respective directories or they can't be used.

4.1.3 Post Installation Steps

After the installation of the application on the Macintosh, some additional steps have to be taken to integrate the application into the operating system. The file types, creator and icons used by the MacLWB have to be registered with the MacOS. This step makes it then possible to recognize LWB files by their icons and to start the LWB by double clicking on one of its files.

The LWB internally uses a variable to access its auxiliary files. This variable contains the position of the directory the LWB was installed in. The installation cannot set this directory in any way. On the Macintosh this is impossible because the Macintosh does not have a concept of environment variables. On Unix, a simple compressed archive does not allow to set any variables. Thus, the variable has to be set at startup of the LWB (see below).

4.2 Startup

As mentioned above, several additional tasks have to be carried through each time the LWB is started. Some of these tasks are necessary because the installation cannot do them, others are just better done while starting the LWB to make sure all things are set correctly each time.

[1] these are the environment variable `LD_Library_Path.` for dynamic libraries and the X font path for the fonts

4.2.1 Configuration Files

The LWB has to make sure that the configuration settings—given by the user the last time the LWB was used—are restored when the LWB is started. That way, the user can adjust the look and feel of the LWB to his liking in a persistent way. Thus, at startup, the LWB has to locate and read the configuration settings from the appropriate file.

4.2.2 Scripting

There has to be a way to allow some sort of scripting of the LWB. This means the capability of executing a predefined set of commands. This can be used to automatically compute results or to compute various problems without user interaction. This has to be taken care of at startup, to allow running the LWB without user interaction.

4.2.3 Dynamic Libraries (Unix)

The LWB on Unix systems makes heavy use of dynamic libraries.[2] The operating system automatically loads all dynamic libraries whenever they are used. This can only be done when the operating system knows where to look for these files. Thus, at startup, the LWB has to make sure that the operating system will look at the right place to find the libraries.[3]

4.2.4 Home Directory

The home directory where the LWB was installed, has to be set at startup. Together with a fixed directory structure, the LWB is then able to locate its auxiliary files, like the font translation files (see 4.3 below), in one of its subdirectories.

4.2.5 Font Directory (Unix)

The LWB supports special fonts containing special symbols for the logical operators used by the LWB. On Unix, X WINDOWS has to be informed where these fonts can be found. Because this cannot be done while installing the LWB, this has to be done at each startup.[4]

[2] the MacLWB on the other hand does not use dynamic libraries at all.

[3] actually, this has to be done *before* the LWB executable is actually started; just starting the program might already cause the operating system to try to load dynamic libraries.

[4] on the Macintosh, fonts are put in the correct directories by the installation.

4.3 Fonts

The additional fonts that are part of an LWB installation have to be made available to the operating system, either by installing them in the correct place or by adjusting the environment at startup.

But this is not all that is necessary to be able to use these fonts. The LWB also has to do a correct mapping of the special symbols, i.e. the correct symbols have to be displayed whenever a special logical operator is used. This has to be done the same way for all operating systems, to prevent that central parts of the LWB, like the parser, have to be adjusted when porting the LWB to a new system.

4.4 Compilation

A special part of the LWB, which is not actually visible in the final product, is the creation of the executable program itself. The compilation process itself should also fulfill some requirements to make current and future management of the LWB easier.

4.4.1 Handling and Speed

The compilation process itself should be as easy as possible, while, at the same time, being as fast as possible. The first requirement makes sure that a recompilation of the LWB can be done without having to read a lot of documentation. The latter ensures that—while developing the LWB—small changes can be integrated into the executable in short time. This keeps the turnaround cycle short and speeds up development, especially for debugging and testing, because small changes can be incorporated into an executable program in short time, allowing immediate testing and, if necessary, debugging.

When building an application, it is not necessary to always recompile all sources. It is much faster to just compile those files that were changed since the last compilation. Unfortunately, some source files depend on other files and have to be recompiled when one of those files is changed, even if it itself was not. The building process has to include this dependency information and use it when compiling. To make management easier, this dependency information should be generated automatically, if possible.

4.4.2 Multi-User Development

A program with the size of the Logics Workbench cannot be developed by a single person. Thus, multiple users have to work on the same program. Therefore, the development process, including editing, compiling and building has to support multiple users.

There are several issues that need to be treated. First of all, every developer needs access to the same set of source files. It should also be possible for each developer to make changes to the files and to distribute these changes to other developers. Because people make mistakes, it should be possible to take back any changes if problems arise. All these demands are comfortably solved by tools like RCS (cf. 3.2.10) or CVS (cf. 3.2.11). The building process just has to handle the automatic interaction with these tools to make sure the newest revision of a files is used for building.

4.4.3 Space Consideration (Unix)

When development for the Logics Workbench was started, hard disk space on the systems was limited.[5] Thus, the compiling and building process had to make sure that only as much disk space is used as really necessary. As long as possible, disk space should be shared between different developers, reusing as much space as possible.

4.4.4 Package Generation (Unix)

The building process should contain the generation of the installation package as well.[6] On Unix systems, the installation packages should be generated using the same procedures as those used for building the program itself. Especially the packages should be generated automatically, without user intervention.

4.4.5 Testing (Unix)

With every change of or addition to the source code of a program, errors may be incorporated as well. Either, code that worked without problems may now be faulty or the newly introduced features don't work as expected.

To detect as much errors as possible, the program should be tested periodically. These tests should be done totally automatically. That way, tests can be carried through without user intervention, especially convenient for time consuming tests. Furthermore, test results should be easily readable and should help to identify the errors as fast as possible.

Because a lot of scripting facilities are only available on Unix systems, testing was only done on such systems. Because the code of the main algorithms, like the provers, is the same for all versions, testing on a specific system is enough to make sure that they work as intended on all systems.

[5] this changed when more disk space became available later.

[6] this is not possible on the Macintosh due to a mouse and graphical oriented package generator and building process.

4.4.6 Documentation (Unix)

As with package generation and testing, the documentation has to be generated automatically as well. This includes the generation of a printable version of the documentation as well as generating the necessary web pages and text files for the LWB help system.

Of course, it is not possible to actually write the documentation automatically. But to make maintenance feasible, a single document should contain all documentation information. Specific types of documentation, like ASCII or HTML, have to be generated automatically from this central document. This generating process should also be initiated and supervised by the general building process of the Logics Workbench.

Because only the building process on Unix systems supports arbitrary commands to be invoked, the documentation is only generated on Unix system and is then copied onto the Macintosh when finished.[7]

4.4.7 Maintenance

Additionally to all the building tasks mentioned above, the building process should also support additional tasks used for program and source code maintenance. This includes cleaning up the directory tree, making backups or preparing the development environment, for example.

4.5 User Interface

The Logics Workbench provides several different user interfaces. These user interfaces are first predetermined by the operating systems, but there is also a distinction between text oriented and graphical user interfaces.

4.5.1 Text Oriented User Interfaces

The ASCII-LWB contains a text oriented user interface without graphical interaction. Instead, it uses simple text input and output to take commands and give back results.

This interface is the most simple and requires the least programming effort. On the other hand, it is not as comfortable for the user as the other interfaces. Nevertheless, it can be used for big benefits with the automatic execution of commands in the LWB. This can be used for automatic testing as well as for automatically computing complex problems.

This type of interface is the easiest to port to another operating system. If an operating system provides a Unix like shell with support for C++ input and output, the interface is already ported because of the C++ standard.

[7] the generated documentation uses a standard format that is usable on all systems.

4.5.2 Graphical User Interfaces

Because text oriented user interfaces are not very comfortable to use, the LWB is also available with a graphical user interface. The implementation of a graphical user interface is much more time consuming and complicated than that of a text oriented one. Furthermore, porting a graphical user interface is much harder as well. Actually, directly porting the user interface is rarely possible and also rarely makes sense. Instead, the user interface has to be completely rewritten. This makes sure that the look and feel of the operating system is considered and also ensure smooth operating on the destination system.

Motif/X

The MOTIF/X interface is the graphical user interface available for Unix systems. It uses MOTIF and X WINDOWS to display its interface and provides various features to make working with the LWB more comfortable. It was created using MOTIFATION (cf. 3.2.7).

This version of the interface can be used in addition or as a replacement of the ASCII, which is also available for Unix systems.

Macintosh

The Macintosh version of the user interface, along with other modifications of the program, makes it available to run the LWB on the Macintosh. Because the Macintosh operating system does not support a shell, using the ASCII version of the LWB is not possible. Thus this interface is the only one available on the Macintosh. Details concerning the implementation of this interface can be found in chapter 6.

Interaction

To make maintenance of the LWB easier, as much of the source code as possible should be shared between the different versions of the LWB. This means sharing of code between different user interfaces, but also sharing between different operating systems.

To make this possible, the interface specific[8] code has to be decoupled from the code of the rest of the LWB. This allows switching the interface without forcing changes in the rest of the LWB code.[9]

[8] and for the reuse of operating system dependent code the system specific.

[9] Unfortunately, the already existing version of the LWB did not already have such a decoupling and introducing it would have required heavy reengineering of much of the LWB; Because this would have required too much work, a complete decoupling could not be done.

4.6 File Handling

The LWB is capable of reading and writing files, mainly session and configuration files. As mentioned in section 2.5, file handling is a system specific task. Nevertheless, certain things should be shared between the different LWB versions.

Because the Macintosh uses an operating system specific system of storing configuration values, configuration files of the Macintosh cannot be used with other systems. Nevertheless, the MacLWB should be able to handle the shell based startup scripts used by the ASCII Version of the LWB.

Session files[10] should be compatible between all systems. Thus, a session file from an XLWB session can be used with the MacLWB as well.

4.7 System Specifics

Several aspects of the LWB code can only be implemented using system specific code. The following lists the system specific parts of the LWB. These parts have to be implemented in a way that allows their exchange and adaptation without interfering with the rest of the code.

4.7.1 Interrupts

Interrupt handling is highly system specific. Not only the name and calling conventions of the functions differ for the various operating systems, but the interrupt handling in general can be totally different. While Unix supports the standard C interrupt handling, the Macintosh does not really have an interrupt handling at all.

Thus, besides interrupt handling, there has to be another way of dealing with user interrupts to allow the same features on all supported systems (see the following section).

4.7.2 Periodical Tasks

Mostly because the Macintosh does not really support interrupts, periodical tasks have to be introduced. This is operating system specific source code that is called at periodical times through program execution. This allows the different operating systems to carry through all tasks that the system has to complete while running a program.

Furthermore, this easily allowed the introduction of the `limitstart()` and `limitstop()` commands, which allow to limit the maximal computation time that may be used by a computation, forcing a break in execution if the time is surpassed.

[10] these are files containing all the text that was entered into the LWB, along with all results that were returned.

4.7.3 Timing

Timing is operating system specific as well. Unlike the interrupt handling, timing issues don't vary too much from system to system. Thus, only calling conventions and the names of the functions differ from system to system. Furthermore, the number of clock ticks fitting into a single second vary.

Timing is used for the `limitstart()` and `limitstop()` commands mentioned above but also for the timing commands used to determine the execution time of one or multiple LWB commands (`timestart()` and `timestop()`).

4.8 Additional Parts

The LWB contains other important parts, which have to be considered when porting. Different to the parts mentioned above, the following parts don't require much changes when porting. It is normally enough to make sure that these parts can be compiled on the target system. Other problems rarely arise. The parts listed below are briefly described to give a complete overview of the LWB. Additional information can be found in [43] or [29].

4.8.1 Modules

The modules encompass the most important part of the Logics Workbench. They contain the various algorithms for the different logics to actually compute results. While other parts of the LWB provide functionality for user interaction or general data structures, the modules contain the real algorithms, like the automatic theorem provers.

Because the implementation of the various modules only use standard C++ code, they don't have to be changed much for different operating systems. What has to be adjusted are mainly compiler specific issues, i.e. things that are valid for one compiler but not allowed in another one.

4.8.2 Parser

The parser is the part of the LWB that gets the user input and prepares them for execution. It takes the user input and splits it into commands and expressions. Then it uses functions and data structures of the kernel and the modules to execute the commands and compute the results. The parser also contains the LWB programming language with its debugger.

Because no system specific parts are contained in the parser, it does not have to be changed when porting it to another operating system. The only problematic part is the automatically generated parser using LEX (cf. 3.2.5) and YACC (cf. 3.2.6). Because these tools are not available on all systems, the generated code has to be copied to other systems instead of individually generating the parser on each system.

4.8.3 Kernel

The kernel contains the general parts of the Logics Workbench that are used by all modules and other parts of the LWB. It contains the basic data structures used for handling formulas and expressions, as well as the internal structures necessary to handle all user and system defined functions and expressions.

The kernel should not contain any system specific code, instead using these parts from the user interface or other system specific parts of the LWB. This way, the kernel can be ported to other systems as easy as the modules or the parser.

Chapter 5

Porting and Maintenance Steps for the LWB

This chapter describes some of the steps that were done to create the Macintosh version of the Logics Workbench. Several of the steps detailed below allow any program to be more portable, sometimes even without a focus on a specific operating system. Additionally, some steps listed below are not necessary for porting but generally make the maintenance of a program and its development environment easier.

This chapter only contains a rough overview of the implementation of the steps and mechanisms involved in porting the LWB and of ensuring its maintenance. More detailed information about the implementations of the graphical user interface of the MacLWB is presented in the next chapter. Additional details to the steps below, can of course be found in the source code of the parts concerned.

5.1 Prerequisites

When the implementation of the MacLWB was started, a Unix Version of the LWB was already finished, including a prototype version of the XLWB with a graphical user interface. Thus, most of the implementations necessary for the Unix version of the LWB were already done. These things needed to be ported in order to create the MacLWB.

The LWB was only running on SOLARIS systems, though. Furthermore, only the SUN CC compiler was used to compile the program. To make transition to the Macintosh easier and to additionally support the LINUX operating system the, Unix version had to be changed as well. This chapter contains information about these changes and also general information to be considered when creating a program that should run on different Unix systems.

The chapter uses the different parts of the LWB introduced in the previous chapter as a guideline to describe which steps were done to make the LWB more portable and ease maintenance, and finally to create the Macintosh and Linux versions. It also contains some general steps that were done to make development easier and to ensure future maintenance of the LWB.

5.2 Modularization

To make the internal structure or larger programs clearer, they are often separated into different modules. These modules are more or less independent from each other and thus make them easier to oversee and understand, mainly because only smaller parts have to be studied at a time.

The same technique can be used to make a program more portable. Instead of mixing system specific code with the rest of the program, it is put into on or more modules. This allows to clearly ascertain which parts need to be adjusted.

The LWB already used modularization for the kernel, parser, and of course for the modules themselves. System specific code on the other hand, was spread over the whole program, and thus needed to be collected and grouped together.

5.2.1 Strategies

Before actually being able to start to extract the system specific parts, they had to be identified. The previous chapter lists mainly two kinds of system specific code. One is the complete user interface. As mentioned, user interfaces are generally hard or impossible to port to another operating system. Thus, they are clearly system specific and need to be separated from the rest. The other is code includes code that requires system specific functions or data structures. This includes the interrupt handling, timing and the periodical tasks.

When the problematic parts of the code are identified, there are several possible ways to separate them from the rest of the program.

Complete Detachment

As part of the system specific code, the user interface can be completely detached from the rest of the program. Such a detachment actually splits a program into two new programs, one containing the user interface and the other the rest of the original program.

A complete detachment of the user interface from the computation part of the program brings several benefits. Foremost, the part containing the non-interface part is most certainly quite easy to port. Furthermore, it allows multiple, distributed computation parts to be used with a single interface. This includes the use of the program via a network connection.[1]

[1] the programs for mathematical symbolic computations MATHEMATICA and MAPLE both support multiple operating systems and in their newer version both make use of complete detachment.

A complete detachment also has drawbacks, though. Foremost, implementing a detached user interface requires a lot of work. The detachment requires the implementation of a communication interface to allow the program to exchange the necessary data, if necessary over some sort of network. The interface is likely to be system specific, and thus may itself porting more complicated. Furthermore, this solution is hard to incorporate into an existing program, because a lot of redesign would be required. It is much better to actually use complete detachment when development is started.

For the LWB, the situation is even worse. The parser and the symbol table of the LWB would have to be incorporated either into the interface or the computation part. Both solutions immediately bring up serious problems. If parser and symbol table are a part of the computation part, the client/server use or the use of multiple computation parts are no more possible, thus removing one of the main advantage of a complete detachment. If they are part of the user interface, then a big part of the LWB kernel has to be present in both, the user interface and the computation part. Furthermore, kernel and parser would have to be heavily redesigned.

For these reasons, a complete detachment was no implemented. Instead, a simpler, more restricted solution was used (see below).

5.2.2 Abstract Interface

Instead of completely separating the user interface from the rest of a program, it can be conceptually separated by using interface functions. The same can be done for all system specific parts, not just the user interface. Thus, it stays a part of the same program but is clearly separated from the rest of the code. These modules, containing user interface and system specific code, are independent from the rest of the program. They communicate with the rest of the program by using special interface functions, grouped together in the abstract interface.

The abstract interface remains the same even if some system specific parts are changed, changing the system specific modules does not require to change any code in the rest of the program. This makes porting much easier and additionally prevents the introduction of new bugs into already existing code. Furthermore, it allows to clearly detect system specific parts, because they are all collected into specific modules. Thus, when the program is ported, only these modules need to be specially treated.

5.2.3 Technical Aspects

Technically, separation using an abstract interface is done by putting all methods with system specific code into a special class. The same class declaration is used for all versions of the program, while its implementation differs from version to version. This easily ensures that the interface remains the same for all implementations, while allowing different implementations for different versions. The building process has to make sure that the correct implementation is

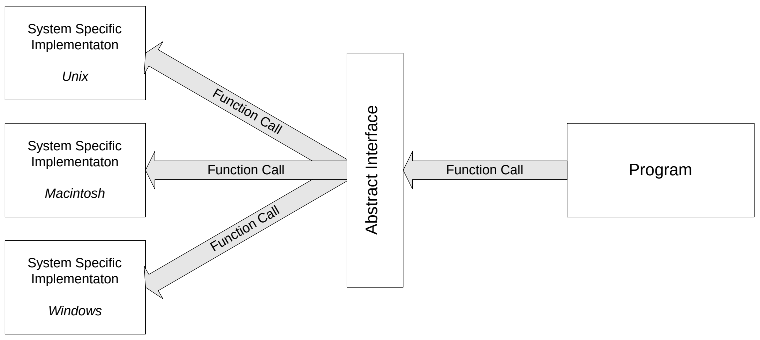

System Specific Implementaton Unix
System Specific Implementaton Macintosh
System Specific Implementaton Windows
Function Call
Function Call
Function Call
Abstract Interface
Function Call
Program

Figure 5.1: Schemata of an abstract interface

used for a specific system. The abstract interface corresponds to the bridge or adaptor patterns mentioned in [19].

Figure 5.1 shows the relationship between different parts of the program and different operating systems.

5.2.4 LWB Implementation

User Interface

The Logics Workbench uses the class `interface` as interface between kernel and parser and the user interface. The class only contains the interface functions, additional data is not necessary.

The implementations of the member functions of the `interface` class are distributed over different files. The file `interfaces/basic/interfaceToLWB.cpp` contains the implementation of the interface functions called from the user interface. It would be possible to directly call functions in the computation part of the LWB. But by using interface functions, the computation part is separated from the user interface and thus changes in the computation part don't require changes in the various user interfaces. Because there are multiple implementations of the user interface, such a change would otherwise require changes in all the implementations for the different systems.

The implementations of the member functions of the `interface` class for the functions called from the computation part are stored in different files, depending on the interface and operating system to be used. For the Macintosh, the implementation is stored in `interfaces/PowerPlant/interfaceToGUI.cpp`. The Unix implementation for the XLWB is stored in `interfaces/Motifation/interfaceToGUI.cpp` and the one for the ASCII LWB is stored in the file `interfaces/ASCII/interfaceToGUI.cpp`.

The building process automatically compiles and links the desired implementation to the final program created. Only one of the implementations is present in the final program, thus no additional steps are necessary to determine which functions to call at runtime.

```
unsigned long
systime::clockticks()
{
  return clock();
}
```

Table 5.1: example implementation of `clockticks()` for the Macintosh

```
unsigned long
systime::clockticks()
{
  static tms procTime;

  times(&procTime);
  return procTime.tms_utime;
}
```

Table 5.2: example implementation of `clockticks()` for Unix

System Specifics

The system specific parts are put into several classes and thus also use multiple abstract interfaces. The interrupt handling is access using the `interrupt` class, the timing functions use the `systime`[2] class, the periodical tasks the `periodical` class, and the process handling for external program in the `sysproc` class.

As with the user interface, the declaration of the class is the same for all systems (in `systems/generic/interrupt.h` etc.) while the implementation of the member functions is different for the Macintosh and for Unix. The implementations are thus stored in the system specific directories `interfaces/PowerMac` and `interfaces/unix`.

Analogous to the user interface, the building process only compiles and links exactly one version of the implementations to the final program.

Example 1 ***Implementation of System Specific Functions***
As an example for the implementation of a system specific function using abstract classes, we look at the method `clockticks()`. This function determines the number of clock ticks that passed so far. This information is used to compute the time required by the LWB commands `timestart()`/`timestop()` and `limitstart()`/`limitstop()`.

The implementation of these functions for the Macintosh and for Unix is shown in tables 5.1 and 5.2. As can be seen, the interface is the same for both functions. Thus, somewhere in the computation part of the LWB, the function call `clockticks()` calls the correct system

[2] the name `time` is already used for system specific purposes, as well as the name `process`.

dependent function and returns the result. The calling of the function does not change, even if the implementation changes.[3]

5.3 Building and Compiling

The building process had to be changed for several reasons. First of all, due to the fact that more developers were starting to make changes to the LWB, a revision control system had to be added. Furthermore, due to space considerations, the makefile had to support a global and a local version. Additionally, some tasks were added to the building process to make overall maintenance easier.

5.3.1 General

The general goal of the building and compiling process is the creation of the executable program which can then be run. Depending on the development environment used, this can be done in different ways. When a graphical development environment is used, then all the details of the building process are done by the development environment and need not concern the developer. All that is required is to define the building and compilation options once and to make sure that all required files are included in the building process.

If a text oriented development environment is used, then the building and compiling process has to be organized in some way. While this requires additional work, it also allows to include additional tasks into the building process, as mentioned in the previous chapter. The rest of this section only deals with the steps that were done for the text oriented building and compiling process used for the LWB on Unix systems.

5.3.2 Local and Global Version

Space requirements for the sources and auxiliary files of the Logic Workbench are quite high, especially if all files have to be stored multiple times because several developers work on the program. To solve this, a distinction was made between a local and a global version of the LWB development.

The local version contains all the source directories in which a developer wants to make changes. Parts of the LWB that are not changed are taken from the global version. Thus, the local version is different for each developer, while the global version is the same for all.

[3] unless the interface is changed, which would require the adjustment of all implementations for all systems as well as all parts actually calling the functions concerned.

The global version on the other hand, contains all sources available. The global version is used to provide the files that are not part of the local version, but also to provide a complete, stable version of the final program. The global version is used in the end to create the final product.

Space

The distinction into a local and global version clearly saves disk space. Each developer only needs to have those files locally which he intends to change, instead of copying all files. It nevertheless allows each developer to change any part of the final program without interfering with other developers.

Speed

Speed is another advantage of the local and global versions. The compilation of global parts of the LWB has to be done only once for all developers. Thus, overall compilation time can be drastically reduced, again without disadvantages.

Implementation

The building process with MAKE does not need much changes for a local and global version. Instead, the set up of the development environment has to be done more complicated. It has to make sure that the local parts are correctly set up and that special links are generated to be able to automatically use the global parts without interfering with the general building and compilation process. All the set up is done with the MAKE target `prepare`.

5.3.3 Configuration

Because development environments tend to change over time, the building process needs a way allowing it to be configured easily. Additionally, the configuration has to support individual changes for specific users without interfering with the configuration of other users. This allows easy adaptation of the building process to the needs of single users without influencing the general building process of other users. The configuration must also support the global and local versions mentioned above. This includes the definition of which parts should be made local and which ones should be take from the global version.

Implementation

The LWB on Unix systems uses the MAKE tool to manage the building process (cf. 3.2.3). MAKE bases the whole building process on makefiles, text files containing rules and variables defining how a program is built or another tasks is carried through. All configurable parameters of the building process are stored in variables. These variables are used for building instead of directly using some specific values, thus allowing easy management of the whole process.

Because the LWB uses many different directories, each requiring an own `makefile`, these configuration variables have to be collected in a single external file. This is done in the file `Makefile.default`, which contains the default configuration for all users, and all directories.

As mentioned, each individual developer should be able to modify his personal configuration without interfering with the default configuration. For that reason, after reading the default Makefile, the file `Makefile.local` is read. This file may set any of the variables of the configuration. Because MAKE only uses the most recent definition of a variable, variables from `Makefile.local` always override the ones from the default configuration.

Additionally, the building process has to be able to differentiate between different versions to build. This is done by automatically use a version specific `Makefile.local` file, which contains all version specific settings, including which interface to build and which implementation of the abstract interfaces to use.

5.3.4 Dependencies

When building a program, the building process has to decide, which files need to be recompiled and for which files the compilation results of the last compilation can be reused. Although it is always possible to just make a rebuild of all files, this would require much more compilation time.[4]

For these reasons, the building process needs a way to find out which files are required to recompile. Clearly, all files that changed since the last compilation have to be compiled anew. But because C/C++ allows the inclusion of other files, a file must also be recompiled in one of the included files was changed. Thus, to check if a file needs to be compiled, not only the file itself has to be checked for modification, but also all included files, the so called dependencies.

To be able to do this, the building process needs to know all the dependencies of a source file. This information can either be entered manually or generated automatically. An automatic generation of the dependency structure is not without problems but is much safer, because changes in the dependency structure are taken into account when compiling.[5]

[4] when first developing for the LWB, compilation of all files of the LWB required approximately 30 minutes; with the faster computer today it still takes about 5 minutes; too much time to wait for a small change.

[5] if the dependencies are not up to date, a file that actually needs compilation might be left out, possibly resulting in a faulty program.

Implementation

Automatically generating all dependencies each time compilation is done was too time consuming, even if only the dependencies are recalculated for changed files. Thus, the generation of the dependencies has to be initiated by hand, using a predefined MAKE target[6]. MAKE then uses the compiler to generate the set of files included from a file and stored them in a special dependency file (`makefile.dep`), which will be automatically used for future compilations. The target `depend_upd` is provided to store the current set of dependencies into the global installation.

Initiating the generating the dependency information by hand is also necessary to make sure that the dependency information is up to date in respect with the global installation. If one user adds a dependency to the global version, all other users need to take care of that automatically. If the dependencies are generated automatically, it could be possible that this global information is lost.

5.3.5 Multi-User Support

As mentioned in 4.4.2, revision control is required for every larger development project. There are several tools available to do the revision control, namely RCS and CVS. The building and compilation process has to work with these tools to always use the correct set of source files.

For the LWB, RCS was selected as revision control system, mainly because CVS was not yet available and the network support provided by CVS was not required because all users have direct access to the same repository.

Using RCS allows to automatically fetch the newest revision of a file when compiling, even if the file was changed by another user. Thus, it automatically makes sure that the newest files are used.[7]

Preparation

As mentioned, RCS requires access to a common repository for all developers. This can easily solved by using file system links to access the global repository. These links need to be set up for each developer. This process is also integrated in the building process with MAKE by providing the target `prepare` which sets up the complete development environment for a user. The includes the correct creation of links to the RCS repository.

[6] the target `depend`, initiated with `make depend`

[7] this is not possible with CVS, where new files have to be fetched from the repository by hand; although this is more complicated, it has the advantage of better control of which files are compiled, because no files are automatically copied.

5.3.6 Documentation

As with other development tasks, it is essential that the program documentation is generated automatically. Thus, the building process has to include rules and commands to generate the documentation as well. For the MAKE tool, documentation can be treated just like source files, thus it is no problem to integrate the documentation into the general compilation and building process.

The documentation is automatically generated from LaTeX sources using various PERL scripts written by Alain Heuerding. The PERL scripts translate the LaTeX sources into several ASCII and HSC files. The HSC files are then used to automatically generate the final HTML files. This intermediate step makes it possible to easily change the layout of the pages without having to change the translation scripts. At the same time, all the pages remain static, allowing an installation of the documentation anywhere desired.[8]

5.3.7 Installation

The creation of the final installation packages is done in several ways. On the Macintosh, the INSTALLER MAKER is used and Linux uses RPM. Both tools cannot be integrated into the general building process. The generation of simple compressed archives, though, can be easily integrated in the general building process with MAKE. Thus, the creation of the installation package can be fully automized. The building process provides the target `package` for this purpose.

Additionally, the building process also allows the creation of source packages, containing all source files of the program. Using the building process for this task has the big advantage, that automatically the same files are stored in the source package that are used to generate the final program.

5.3.8 Tests

One of the most important tasks of software development is testing the final application. This task is so important because it helps ensuring that an application works correctly. This not only means that the application runs without crashing but also has to make sure that the computed results are accurate.

For the LWB, this means that all algorithms have to be checked for two criterias. First, the testing has to make sure that the algorithms terminate without crashing or otherwise producing errors. Second, testing has to ensure that the results obtained are correct, thus the algorithm produces the correct result, at least for the examples tested.

[8] the layout change could also be done using server side includes or other dynamic pages, but then the pages would require a specially configured server to be viewed.

It is not easily possible to test user interfaces, especially graphical ones. Thus a test of these features of the program was not done.

Implementation

As the previous task, testing was also implemented by including a special target into the makefiles. The target used is `test`, thus with `make test` testing can be initiated. The actual testing is then done by a perl script, which feeds a special test file (extension `.lwb`) to the LWB executable. The resulting output is compared to a correct, manually computed file (extension `.lwbout`). If the results don't match, a warning message is printed.

Using this test facility, it is quite easy to check that code still works after making changes. On the other hand, the testing requires the creation of test files with the commands to test and also writing and manually computing a file with the correct results.

5.4 Compiler

An important step when creating a potentially portable program is to compile it with different compilers, even if it is on the same system. Different compilers, although supporting the same programming language, generally interpret source code in different, sometimes surprising ways. Code that quietly compiles on one compiler may generate a lot of warnings on another compiler or even produce compilation errors. Most of the time, these problems arise with code that is actually faulty and needs to be corrected but somehow compiled without errors or warnings. While removing those errors and warnings is time consuming, it nevertheless assures that the program is more stable and can be more easily compiled on other systems and with other compilers.

For the LWB, the change to another compiler was required because of the port to Linux. The SUN compiler used so far was not available, thus another compiler had to be used. The change to a new compiler allowed to detect several problematic and even some outright wrong pieces of code.

5.5 Scripts

Several scripts were necessary to ease the installation of the LWB as mentioned in 4.1. The following steps are only necessary for the Unix versions of the LWB and do not concern the Macintosh version. On the Macintosh, the installation does automatically take care of the necessary adjustments during the installation.

5.5.1 Startup Script

To make it possible to install the LWB from a simple compressed archive without requiring several manual steps after the installation, several things have to be done before starting the LWB. That means, instead of adjusting the environment after the installation, the same steps are done each time the application is run. All the things that can be done after the LWB is actually started are done in the program itself. Only steps that need to be done before the application is started are done in the startup script.

The program can be started by simply executing the startup script. The script automatically sets up the environment as necessary and then simply starts the LWB application itself.

The startup script makes the following settings to the environment:

- determine in which directory the dynamic libraries are stored and add the determined value to the environment variable for the dynamic libraries[9],
- check if there is a configuration file to read and give the found filename over to the LWB executable,
- for the XLWB additionally add the directory where the fonts are stored to the X WINDOWS font path[10].

The startup script for the ASCII LWB is shown in figure 5.3. The script for the XLWB additionally just contains the setting of the X font path.

5.5.2 Invocation Scripts for External Programs

Scripts are not only used for calling the LWB itself, but also every time the LWB calls an external program. Currently this is the case for the help system and the ProofWish tool (cf. [4]).

Again, an intermediate step between the LWB and the environment serves to decouple the application from the external programs used. Thus, the invocation details used to launch an external program can be changed without requiring changes of the source code of the application.

The invocation script for the LWB help system is shown in figure 5.4. This default script calls NETSCAPE with the appropriate page to show.

[9] `LD_LIBRARY_PATH`

[10] via `xset`

```
#!/bin/sh
\# variables that need contents
\# $dir:   directory with the path to the executable (may contain
\#         relative paths)
\# $fonts: directory with the path to the fonts (may NOT contain
\#         relative paths)
\# $lib:   directory with the library path of the lwb (may contain
\#         relative paths)
\#

if test -h $0; then
  echo "Sorry, the LWB cannot be started via a symbolic link !"
  exit 1
fi

\# determine the directory in which the current script is located
if test ‘uname‘ = "Linux"; then
  dir=‘dirname $0‘
else
  dir=‘which $0‘
  dir=‘dirname $dir‘
fi

dir=‘/bin/sh -c "cd $dir; pwd"‘
lib=‘/bin/sh -c "cd $dir/../lib; pwd"‘

if test ! -f $dir/lwb.exec; then
  echo "Executable $dir/lwb.exec not found, aborting !"
  exit
fi

# set the LD_LIBRARY_PATH to the corresponding directory
LD_LIBRARY_PATH=$LD_LIBRARY_PATH${LD_LIBRARY_PATH:+:}$lib
export LD_LIBRARY_PATH

echo "Starting the LWB, please wait..."

\# if there is a configuration file, we use that as well
if test -f .lwbrc; then
  $dir/lwb.exec .lwbrc $*
else
  if test -f $dir/.lwbrc; then
    $dir/lwb.exec $dir/.lwbrc $*
  else
    $dir/lwb.exec $*
  fi
fi
```

Table 5.3: LWB startup script

```
#!/bin/sh

netscape -remote "openURL("$1")" 2> /dev/null
result=$?

\# netscape is not yet running
if [ $result -ne 0 ]; then
  netscape $1 &
fi
```

Table 5.4: Launch script for the LWB help system

Chapter 6

MacLWB Implementation

6.1 Introduction

This chapter describes the structure and implementation of the graphical user interface (GUI) of the LWB for the Macintosh (MacLWB). While earlier chapters showed why something has to be done in a special way, this chapter shows how things were really done in the end. This chapter only contains information concerning the Macintosh version of the LWB. While a version for Linux systems was also done, using the strategies described in the previous chapters allowed to implement this version on-the-fly, without much additional considerations.

The tools and utilities described in chapters 3 were heavily used in the creation of the user interface. The METROWERKS CONSTRUCTOR was used to graphically create the positions of the individual interface elements. The underlying code had to be created normally, though, with all the limitations given from the use of the POWER PLANT library structure, as mentioned earlier (cf. 3.1.2).

Some side glances to the graphical user interface of the Unix version will be done, to show both versions are used the same way. At the same time the native look and feel of the Macintosh version is preserved. The versions only differ where the general look and feel of the Macintosh differs with the one of Unix.

The first part of this chapter will deal with the user interface from a users point of view, i.e. it will briefly show how the Logics Workbench on the Macintosh is used and what features the user interface offers (more details are available in the LWB help pages at `http://www.lwb.unibe.ch`). The second part of the chapter takes a closer look at the implementations of the functionality described in the first part.

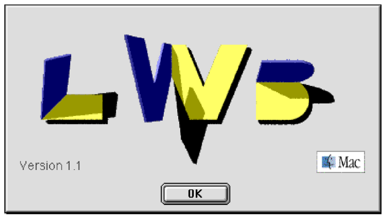
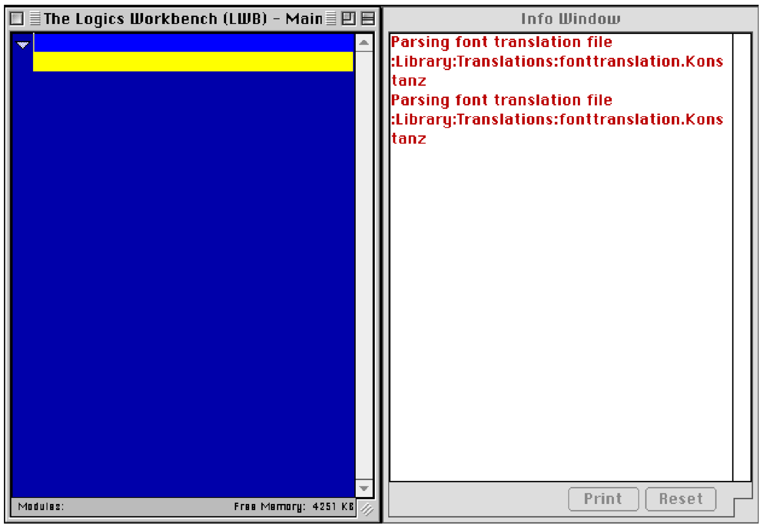


Figure 6.1: MacLWB: startup message



```
Parsing font translation file
:Library:Translations:fonttranslation.Kons
tanz
Parsing font translation file
:Library:Translations:fonttranslation.Kons
tanz
```

Figure 6.2: MacLWB: main and info windows

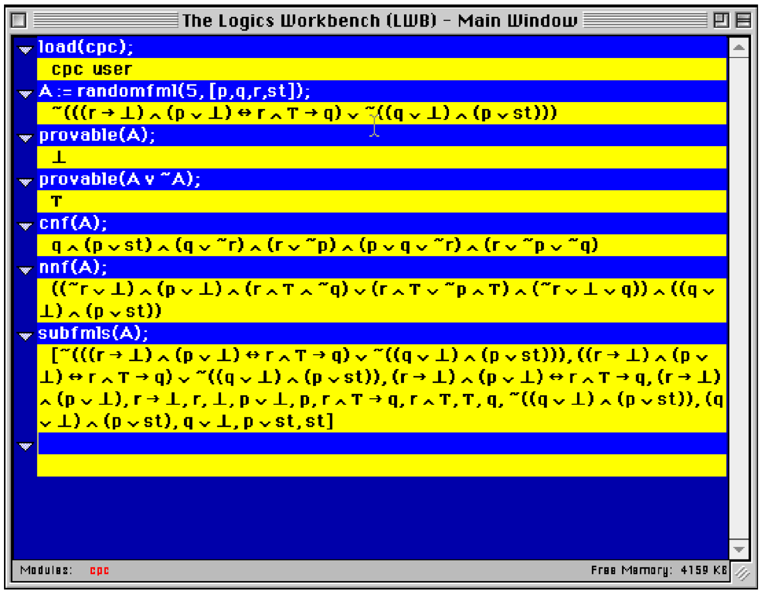
6.2 A User's View

The first thing a user sees when starting the Logics Workbench on the Macintosh, is the about message, a small display showing the version number and logo of the LWB (see figure 6.1). It automatically disappears after a short time, to be replaced by the main and info windows of the LWB (figure 6.2). The LWB is now fully started and ready for user input.

First attention will be focused to the main window, the most important part of the user interface and later to the info window.

6.2.1 The Main Window

The main window, as its name implies, is the central input and output interface of the MacLWB. All user inputs are entered in this window and here all results of the LWB are displayed.



```
load(cpc);
  cpc user
A := randomfml(5, [p,q,r,st]);
  ~(((r → ⊥) ∧ (p ∨ ⊥) ↔ r ∧ T → q) ∨ ~((q ∨ ⊥) ∧ (p ∨ st)))
provable(A);
  ⊥
provable(A v ~A);
  T
cnf(A);
  q ∧ (p ∨ st) ∧ (q ∨ ~r) ∧ (r ∨ ~p) ∧ (p ∨ q ∨ ~r) ∧ (r ∨ ~p ∨ ~q)
nnf(A);
  ((~r ∨ ⊥) ∧ (p ∨ ⊥) ∧ (r ∧ T ∧ ~q) ∨ (r ∧ T ∨ ~p ∧ T) ∧ (~r ∨ ⊥ ∨ q)) ∧ ((q ∨
⊥) ∧ (p ∨ st))
subfmls(A);
  [~(((r → ⊥) ∧ (p ∨ ⊥) ↔ r ∧ T → q) ∨ ~((q ∨ ⊥) ∧ (p ∨ st))), ((r → ⊥) ∧ (p ∨
⊥) ↔ r ∧ T → q) ∨ ~((q ∨ ⊥) ∧ (p ∨ st)), (r → ⊥) ∧ (p ∨ ⊥) ↔ r ∧ T → q, (r → ⊥)
∧ (p ∨ ⊥), r → ⊥, r, ⊥, p ∨ ⊥, p, r ∧ T → q, r ∧ T, T, q, ~((q ∨ ⊥) ∧ (p ∨ st)), (q
∨ ⊥) ∧ (p ∨ st), q ∨ ⊥, p ∨ st, st]

Modules: cpc                                          Free Memory: 4159 KB
```

Figure 6.3: MacLWB: main window after entering some commands

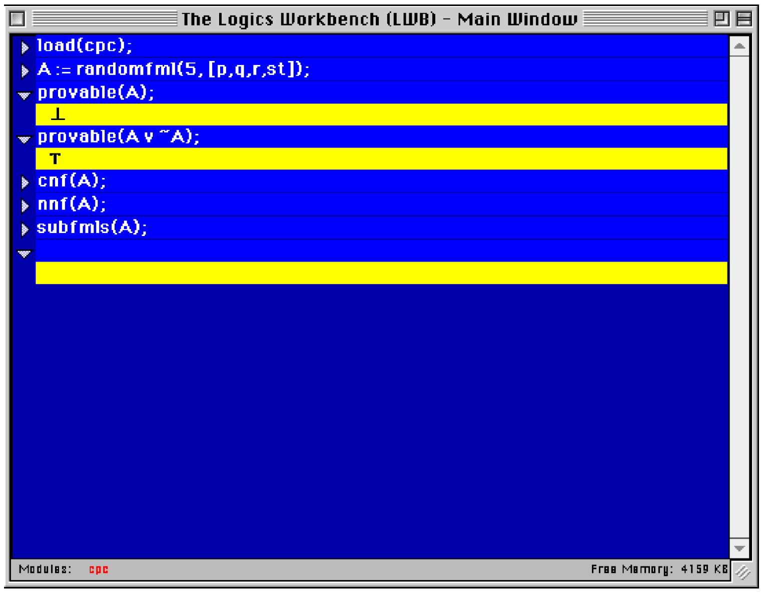
Beside the standard interface elements of a window, like close or resize elements, the main window additionally has three main elements. These are the regions, the module display and the memory display.

Regions

In the beginning, the main window only contains three things, as shown in figure 6.2. The blue bar is the so called input region. It is used to enter formulas and commands. Any text entered here will be given to the LWB parser for interpretation and execution. Right below the input region is its associated output region, shown as a yellow bar. This region will contain the output, i.e. result of the statements given in the input region.

Figure 6.3 shows the same regions after some commands have been entered and the LWB generated some output. As can be seen, the LWB automatically generates a new pair of input and output regions after the first one is completed. Thus, previous in- and outputs are still visible and accessible for copying and pasting. Of course it is possible to navigate in and between these input and output regions using the cursor keys or by selecting a specific text position with the mouse pointer.

Of the three parts mentioned at the beginning of this section we have looked at two so far. What's left is the small triangular button just left of the input region. This button can be used to shrink its associated input and output regions. When the triangle points down, the complete input and output regions are displayed. If the triangle is clicked, it changes to pointing right and now only the first line of the input region and no output region is displayed. Another click on the triangle



```
load(cpc);
A := randomfml(5, [p,q,r,st]);
provable(A);
  ⊥
provable(A v ~A);
  T
cnf(A);
nnf(A);
subfmls(A);
```

```
Modules:   cpc                                    Free Memory: 4159 KB
```

Figure 6.4: MacLWB: main window with some regions hidden

shows the complete input and output region again. This can be used to get a better overview of the whole LWB session by hiding currently not used information. If some sort of indication is present in the first line of the input region to what is stored in that input/output region pair, then a desired input/output region can easily be found afterwards. Figure 6.4 shows the same session as in figure 6.3, but this time with some of the regions hidden.

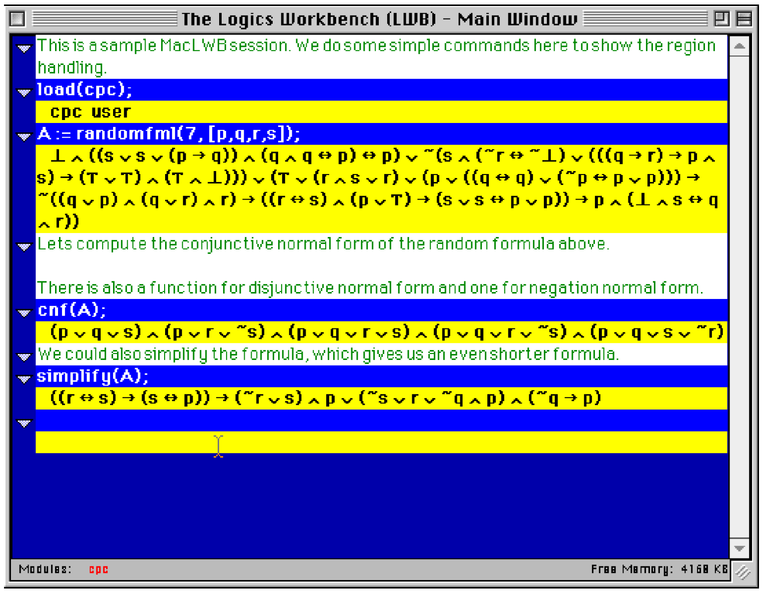
Additionally, there is third type of region available to the user. The so called text region is a region where text can be entered which will not be executed or evaluated at all. Regions of this type are mainly used for documentation. Figure 6.5 shows an example.

As shown in figure 6.5, each type of region can have its own fore- and background color and its own font, all configurable by the user.

Module and Memory Display

On the bottom border of the main window are two displays showing information about the current state of the MacLWB. On the left side is the module display, showing the current order of loaded LWB modules. Functions called are searched in the modules in the order shown. Figure 6.6 shows a MacLWB session with several modules loaded.

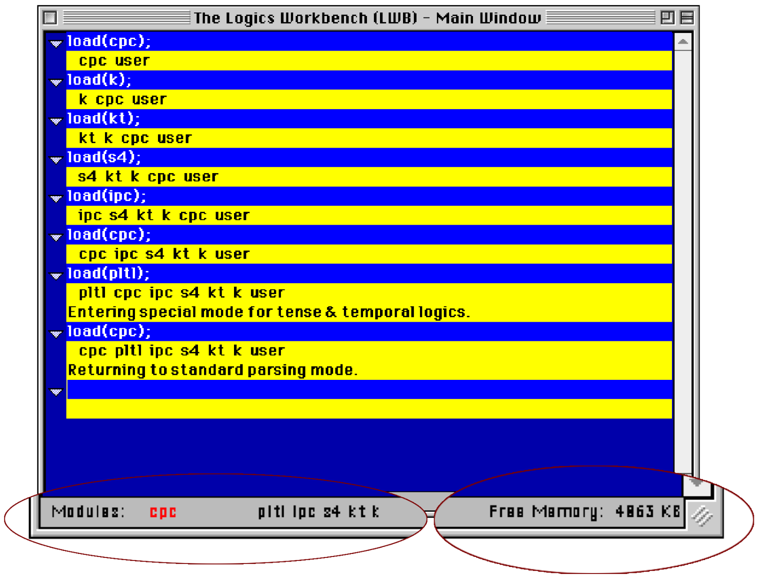
As shown in 2.11.2, the Macintosh provides each application with a set amount of memory as defined in the FINDER. For the MacLWB, it is hard to determine the amount of memory required beforehand, because it highly depends on the kind of computations done. For that reason, it is essential for the user to know how much memory is free for the application to use. This helps



```
The Logics Workbench (LWB) - Main Window
This is a sample MacLWB session. We do some simple commands here to show the region
handling.
load(cpc);
  cpc user
A := randomfml(7, [p,q,r,s]);
  ⊥ ∧ ((s ∨ s ∨ (p → q)) ∧ (q ∧ q ↔ p) ↔ p) ∨ ~(s ∧ (~r ↔ ~⊥) ∨ (((q → r) → p ∧
s) → (T ∨ T) ∧ (T ∧ ⊥))) ∨ (T ∨ (r ∧ s ∨ r) ∨ (p ∨ ((q ↔ q) ∨ (~p ↔ p ∨ p))) →
~((q ∨ p) ∧ (q ∨ r) ∧ r) → ((r ↔ s) ∧ (p ∨ T) → (s ∨ s ↔ p ∨ p)) → p ∧ (⊥ ∧ s ↔ q
∧ r))
Lets compute the conjunctive normal form of the random formula above.

There is also a function for disjunctive normal form and one for negation normal form.
cnf(A);
  (p ∨ q ∨ s) ∧ (p ∨ r ∨ ~s) ∧ (p ∨ q ∨ r ∨ s) ∧ (p ∨ q ∨ r ∨ ~s) ∧ (p ∨ q ∨ s ∨ ~r)
We could also simplify the formula, which gives us an even shorter formula.
simplify(A);
  ((r ↔ s) → (s ↔ p)) → (~r ∨ s) ∧ p ∨ (~s ∨ r ∨ ~q ∧ p) ∧ (~q → p)
Modules: cpc                                                    Free Memory: 4168 KB
```

Figure 6.5: MacLWB: main window with some text regions



```
The Logics Workbench (LWB) - Main Window
load(cpc);
  cpc user
load(k);
  k cpc user
load(kt);
  kt k cpc user
load(s4);
  s4 kt k cpc user
load(ipc);
  ipc s4 kt k cpc user
load(cpc);
  cpc ipc s4 kt k user
load(pltl);
  pltl cpc ipc s4 kt k user
Entering special mode for tense & temporal logics.
load(cpc);
  cpc pltl ipc s4 kt k user
Returning to standard parsing mode.
Modules: cpc         pltl ipc s4 kt k          Free Memory: 4863 KB
```

Figure 6.6: MacLWB: main window with the module and memory displays

preventing low memory situations. Given that information, a user is able to adjust the preset amount of memory in the FINDER if not enough memory seems to be available.

The MacLWB shows the free memory in the bottom right corner of the window border, as shown in figure 6.6. This display is frequently updated to always show the current amount of free memory.

6.2.2 Menus and Keyboard

The following sections describe the various keyboard and menu commands. Some of these commands can either be called using the keyboard or using the menu entry, while others can only be called from the keyboard or a menu, respectively.

The keystrokes and menu commands will only briefly be treated below. The LWB Help Pages at `http://www.lwb.unibe.ch/mac/keyboard.html` contain a complete list and descriptions of all keystrokes available in the MacLWB.

Navigation

As already mentioned, the cursor keys can be used to navigate the current input position, either inside a region, or if a region boundary is reached, between adjoining regions. If at the same time as using some cursor movement, the shift key is pressed, then the text passed over will be selected, to be available for copying or deletion.[1]

Using the command keys, it's possible to directly position the cursor at either the start or the beginning of the line, using the left and right cursor key respectively, or at the beginning or end of the whole region, using the up and down keys. Furthermore, using the option key in addition to the left and right keys allows the hiding and showing of the current input/output region, while the option key in combination with the up and down keys jumps to the beginning of the first or to the end of the last region, respectively.

Region Handling

In addition to the navigation commands listed above, the MacLWB also supports several commands dealing with complete regions and their contents.

First, cut, copy and paste works in all the regions as expected. Thus, it is possible to copy and paste a text from one region to another. Of course it is not possible to change or enter text in an output region, because they cannot be modified. Second, there is a command to insert a new pair of input and output regions before the current one. It's also possible to insert a new text region in the same way. Third, input regions support automatic name completion. After entering the

[1] This feature does not work over region boundaries, though.

beginning of a name of an LWB symbol, pressing the tabulator key will automatically complete the symbol name as long as it is unique, using the current symbol table. This works for predefined functions and variables as well as for user defined ones. Lastly, there is a command available to print a single region.

The return key has a special meaning in an input region. It terminates the commands entered and hands them over to the LWB for parsing and computation. The LWB will compute the result and show it in the corresponding output region, as well as creating a new input/output region pair, if necessary.

Session Handling

Some commands are available to deal with complete sessions, i.e. all currently shown regions.

The reset command resets the whole session, i.e. all input and output regions are deleted and a new, empty pair of input/output regions is created. As is the case with the X version of the LWB, this does not affect the internal state of the LWB itself, i.e. all variables or functions keep their values and are not reset.[2]

The MacLWB also has commands to save and load complete sessions. They only load and save the text in the regions but won't actually change or save the internal state of the LWB. Thus, saving the current session will not store the variables defined or results obtained. Furthermore, reading a session does not define or change any values, just sets the text of the regions. The sessions need to be reevaluated to actually set the values again. This requires the repeated computation of all results.

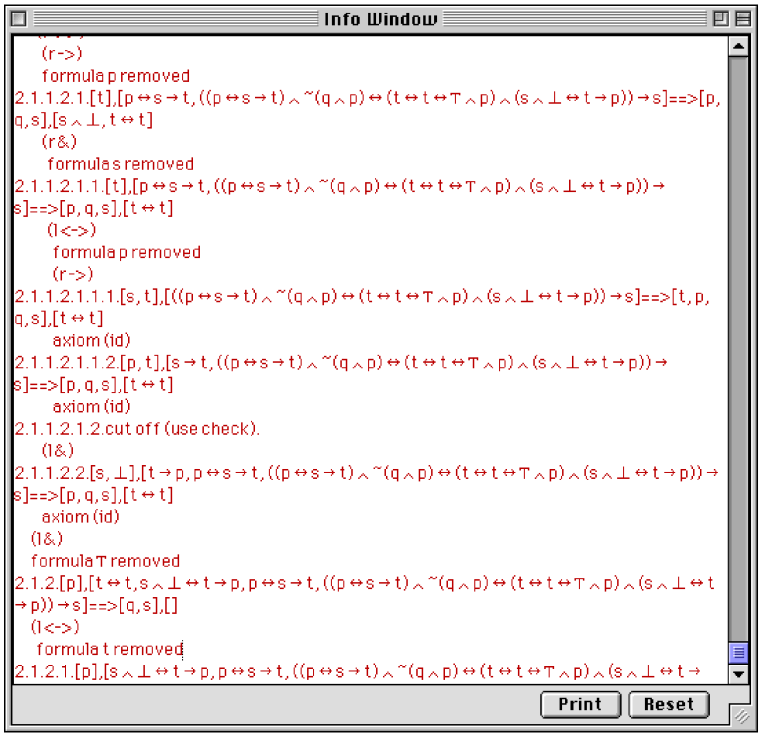
Lastly, there is a command to print all the current regions. The regions will be printed as they are shown, using the same colors and the same font. Another command shows or hides all current regions and one will reevaluate all regions, i.e. recompute the output of all regions starting with the first and ending with the last.

Program Handling

As a last group, there are some commands that affect the whole MacLWB. Of course there is a command to quit the MacLWB. Another commands allows to display the main help page (see below for more information about the help system). While the MacLWB is computing a result, a command is available to interrupt the computation and to return to editing. Of course, in such a case, no result is obtained from the computation, but a computation that takes too much time my be stopped.

Above, a command was mentioned that saves the text of the LWB regions without interfering with the internal state of the LWB. Similarly, a command is available to save or read the internal

[2] to reset the internal state of the LWB, the `reset` command is used; this does not affect the display in any way, though.



```
    (r->)
    formula p removed
2.1.1.2.1.[t],[p↔s→t,((p↔s→t)∧~(q∧p)↔(t↔t↔T∧p)∧(s∧⊥↔t→p))→s]==>[p,
q,s],[s∧⊥,t↔t]
    (r&)
     formula s removed
2.1.1.2.1.1.[t],[p↔s→t,((p↔s→t)∧~(q∧p)↔(t↔t↔T∧p)∧(s∧⊥↔t→p))→
s]==>[p,q,s],[t↔t]
     (l<->)
      formula p removed
      (r->)
2.1.1.2.1.1.1.[s,t],[((p↔s→t)∧~(q∧p)↔(t↔t↔T∧p)∧(s∧⊥↔t→p))→s]==>[t,p,
q,s],[t↔t]
      axiom (id)
2.1.1.2.1.1.2.[p,t],[s→t,((p↔s→t)∧~(q∧p)↔(t↔t↔T∧p)∧(s∧⊥↔t→p))→
s]==>[p,q,s],[t↔t]
      axiom (id)
2.1.1.2.1.2.cut off (use check).
    (l&)
2.1.1.2.2.[s,⊥],[t→p,p↔s→t,((p↔s→t)∧~(q∧p)↔(t↔t↔T∧p)∧(s∧⊥↔t→p))→
s]==>[p,q,s],[t↔t]
    axiom (id)
   (l&)
   formula T removed
2.1.2.[p],[t↔t,s∧⊥↔t→p,p↔s→t,((p↔s→t)∧~(q∧p)↔(t↔t↔T∧p)∧(s∧⊥↔t
→p))→s]==>[q,s],[]
   (l<->)
    formula t removed
2.1.2.1.[p],[s∧⊥↔t→p,p↔s→t,((p↔s→t)∧~(q∧p)↔(t↔t↔T∧p)∧(s∧⊥↔t→
```

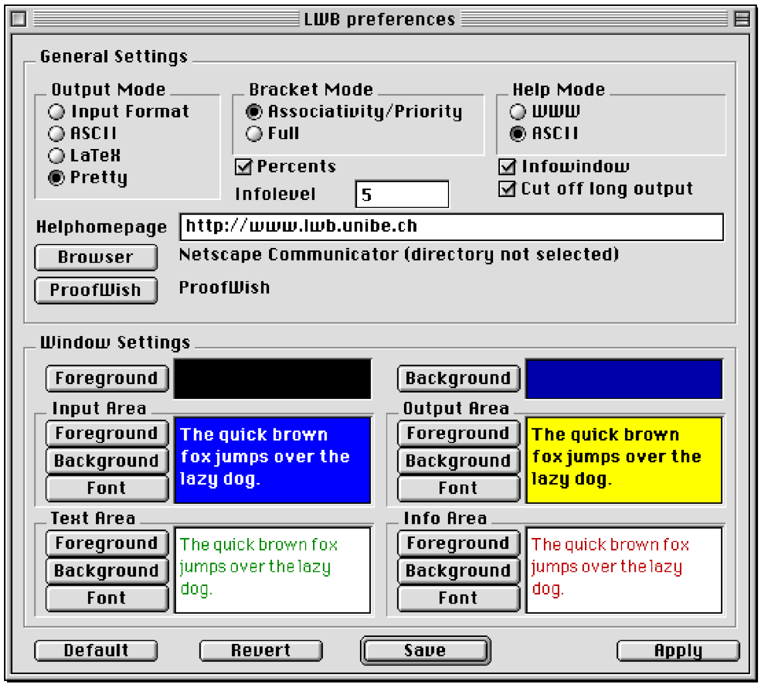
Figure 6.7: MacLWB: info window during a proof

state of the LWB. Commands for reading and writing the current configuration and for setting up the page used for printing are available as well. Additionally, three commands allow to load, unload and change the order of LWB modules. Last but not least, one command shows or hides the info window and one sets the preferences of the program.

6.2.3 Info Window

Next to the main window, another window may be visible while working with the LWB, the info window. This window displays information produced by the LWB algorithms. Output only appears in this window, if the info level in the preferences is set to a value above 0 (see below). The higher the info level setting, the more output will be generated. Especially for proofs, the amount of output can be quite high, and thus slow down computation considerably. The information in the info window can be quite useful to see in more detail what exactly happens when the LWB computes a result. This is especially useful while debugging. Figure 6.7 shows an example info window while making a proof with the infolevel set to 5.

The info window itself has two buttons, one to reset the window, i.e. clear its contents, and the other to print the whole text.



LWB preferences

General Settings

Output Mode: Input Format / ASCII / LaTeX / [✓ Pretty]

Bracket Mode: [✓ Associativity/Priority] / Full

Help Mode: WWW / [✓ ASCII]

✓ Percents

Infolevel 5

✓ Infowindow

✓ Cut off long output

Helphomepage http://www.lwb.unibe.ch

Browser: Netscape Communicator (directory not selected)

ProofWish: ProofWish

Window Settings

Input Area: The quick brown fox jumps over the lazy dog.

Output Area: The quick brown fox jumps over the lazy dog.

Text Area: The quick brown fox jumps over the lazy dog.

Info Area: The quick brown fox jumps over the lazy dog.

Figure 6.8: MacLWB: the preferences dialog

6.2.4 Preferences

The preferences are used to set the LWB configuration variables and to adjust the look of the graphical user interface. First the configuration settings that influence the behavior of the LWB and its algorithms will be explained. Later the ones that can be used to adjust the general outlook of the MacLWB. Figure 6.8 shows the dialog used to set the preference values.

All the configuration settings influencing the LWB itself can either be adjusted with the graphical user interface or by using the `set()` command of the LWB. For a complete overview of these variables look at the LWB home page at `http://www.lwb.unibe.ch`. The values set in the MacLWB preferences are stored in the LWB preferences file and are automatically reloaded upon the next start.

The MacLWB allows to configure the following values:

Output Mode
This value defines how normal output is printed. It can be set to either 'output', printing in ASCII using normal characters for the logical operators, in 'ASCII', using full words for the operators, to 'LaTeX' to print in LaTeXor to 'pretty', using special symbols for the logical connectors.

Bracket Mode
This mode defines if all brackets should be shown or, because of associativity and operator precedence, only the necessary ones.

Help Mode
This mode chooses the type of help system to be used. Help information will either be shown as ASCII text in the info window or as a web page, using the configured web browser.
Percents
This switch determines if a progress bar is shown when computing a result. Computing without a progress bar may be a trifle faster, but does not give any indication how long a computation may take.
Infolevel
This defines the level of output generated in the info window. The higher the number, the more output will be generated.
Cut Off
If this switch is set, then long output is cut off after some amount of text; this can greatly increase computation speed if a lot of output is generated. This switch applies to the text in an output region as well as to the one in the info window.

The following values can only be configured using the preferences panel and not directly with LWB commands. They configure the user interface and not the LWB itself, thus setting them in another version of the LWB would not make sense.

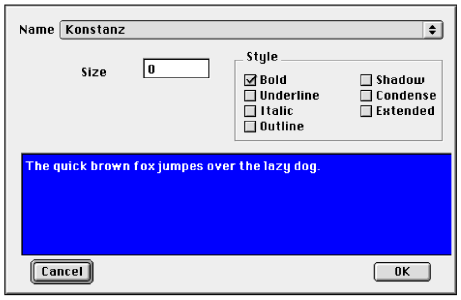
Infowindow
This switch determines if the info window will be shown at startup of the MacLWB or not.
Help Homepage
This defines the main page to be shown for the help system; this can mainly be adjusted if the help files are installed locally.
Browser
This value specifies the program that is used to view the help pages; any browser compatible with the standard Mac events can be used.[3]
ProofWish
This value determines the ProofWish tool used to display classical proofs. This tool is part of the LWB installation, but nevertheless must be selected here.

Finally, there are several options which define the graphical look of the operating system, i.e. its fonts and colors. It is possible to set the general fore- and background color of a window and to set the fore- and background color and font of input, output, text, and info regions individually.

Font Dialog

The font dialog is a sub window, displayed whenever a font has to be selected. It allows the selection of the font, its size and other display styles, like bold or italic printing. An example is depicted in figure 6.9.

[3] at least Netscape Communicator and Internet Explorer are compatible.


Name Konstanz

Size 0

Style

- [✓ Bold]
- Underline
- Italic
- Outline
- Shadow
- Condense
- Extended

The quick brown fox jumpes over the lazy dog.

Figure 6.9: MacLWB: the preferences font dialog

6.2.5 Help System

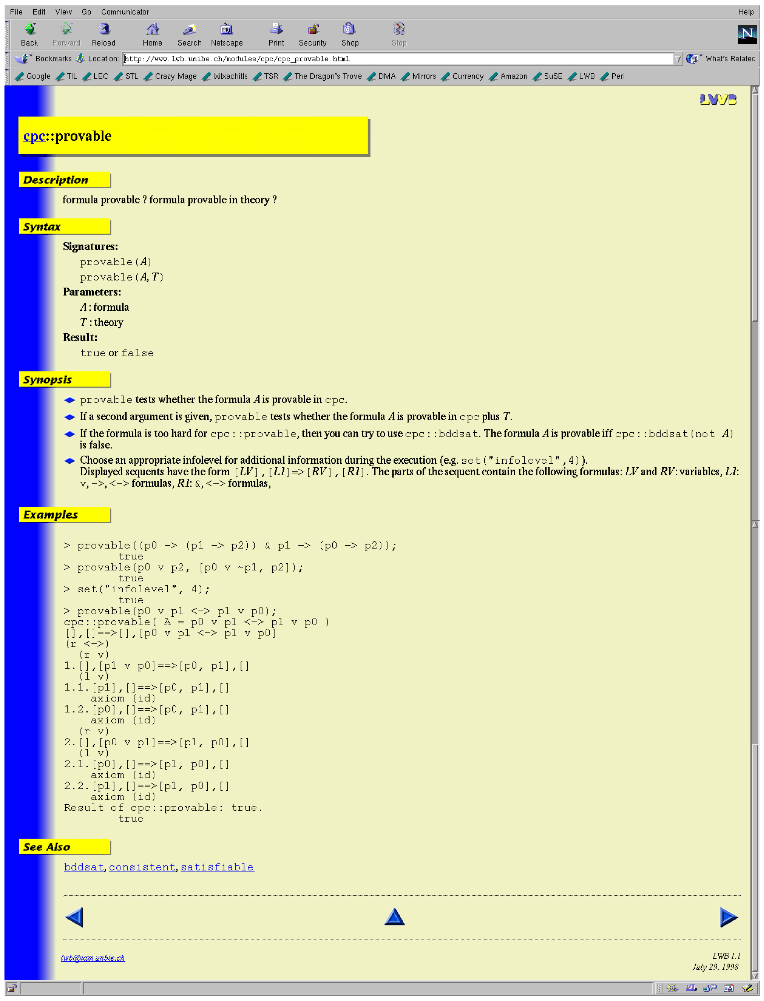
The MacLWB has the same two help systems as the other versions of the LWB, namely an ASCII and a WWW version. The ASCII version consists on several text files containing descriptions for all built-in functions. These texts are shown in the info window whenever the user issues a help command.

Furthermore, there is the web based help system using the LWB web pages. These pages can either be viewed directly from the main LWB page over the Internet or can be installed locally. The former is easier to install and the latter faster to access. The correct page is automatically shown as a reaction to a help request in the LWB. If necessary, the web browser selected in the preferences will be started. Using special Macintosh events, the browser is informed to show the correct page, either from the Internet or from the local disk. The information for both versions of the help texts is taken from the same set of LaTeX files, thus both versions contain the same information.

Figure 6.10 shows an example help page of the WWW help system for the provable command of the classical propositional calculus (cpc). Figure 6.11 shows the same information displayed in the info window with the ASCII help system.

6.2.6 Progress Indicator

The progress indicator is a simple display to show the progress of a computation in the LWB. It shows how much of a computation was already done and thus allows to estimate the time required to finish the computation. Furthermore, it allows the user to stop a computation if it will take too long. Figure 6.12 shows an example.


File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://www.lwb.unibe.ch/modules/cpc/cpc_provable.html What's Related
Google TIL LEO STL Crazy Mage Ixitxachitls TSR The Dragon's Trove DMA Mirrors Currency Amazon SuSE LWB Perl

cpc::provable

Description

formula provable ? formula provable in theory ?

Syntax

Signatures:

`provable(`*A*`)`
`provable(`*A*, *T*`)`

Parameters:

A : formula
T : theory

Result:

`true` or `false`

Synopsis

- `provable` tests whether the formula *A* is provable in `cpc`.
- If a second argument is given, `provable` tests whether the formula *A* is provable in `cpc` plus *T*.
- If the formula is too hard for `cpc::provable`, then you can try to use `cpc::bddsat`. The formula *A* is provable iff `cpc::bddsat(not A)` is false.
- Choose an appropriate infolevel for additional information during the execution (e.g. `set("infolevel",4)`).
  Displayed sequents have the form `[`*LV*`],[`*L1*`]=>[`*RV*`],[`*R1*`]`. The parts of the sequent contain the following formulas: *LV* and *RV*: variables, *L1*: v, ->, <-> formulas, *R1*: &, <-> formulas,

Examples

```
> provable((p0 -> (p1 -> p2)) & p1 -> (p0 -> p2));
        true
> provable(p0 v p2, [p0 v ~p1, p2]);
        true
> set("infolevel", 4);
        true
> provable(p0 v p1 <-> p1 v p0);
cpc::provable( A = p0 v p1 <-> p1 v p0 )
[],[]==>[],[p0 v p1 <-> p1 v p0]
(r <->)
  (r v)
1.[],[p1 v p0]==>[p0, p1],[]
  (l v)
1.1.[p1],[]==>[p0, p1],[]
    axiom (id)
1.2.[p0],[]==>[p0, p1],[]
    axiom (id)
  (r v)
2.[],[p0 v p1]==>[p1, p0],[]
  (l v)
2.1.[p0],[]==>[p1, p0],[]
    axiom (id)
2.2.[p1],[]==>[p1, p0],[]
    axiom (id)
Result of cpc::provable: true.
        true
```

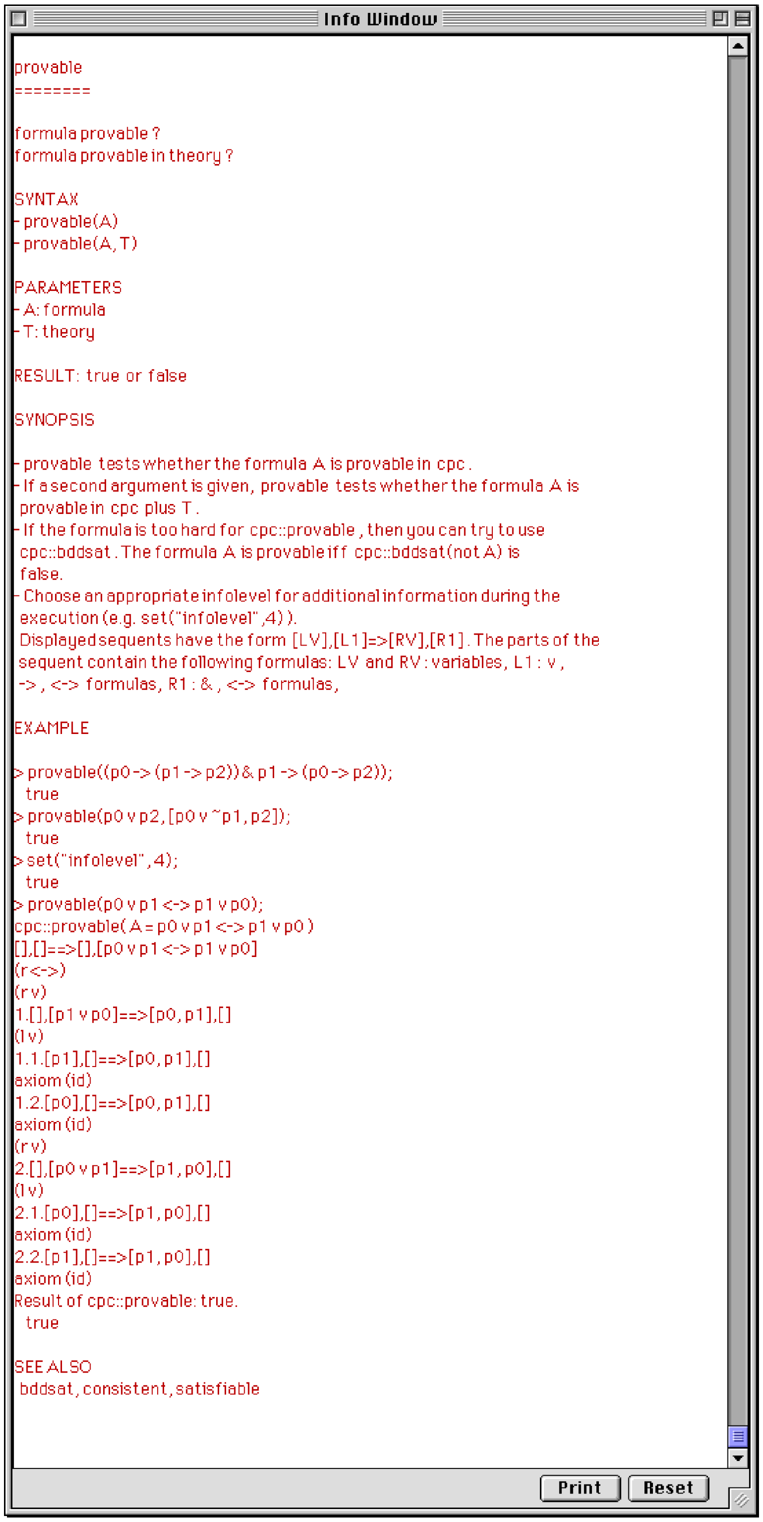
See Also

bddsat, consistent, satisfiable

lwb@iam.unbie.ch

LWB 1.1
July 29, 1998

Figure 6.10: MacLWB: WWW based help system



```
provable
========

formula provable ?
formula provable in theory ?

SYNTAX
- provable(A)
- provable(A, T)

PARAMETERS
- A: formula
- T: theory

RESULT: true or false

SYNOPSIS

- provable  tests whether the formula A is provable in cpc .
- If a second argument is given, provable  tests whether the formula A is
 provable in cpc plus T .
- If the formula is too hard for cpc::provable , then you can try to use
 cpc::bddsat . The formula A is provable iff  cpc::bddsat(not A) is
 false.
- Choose an appropriate infolevel for additional information during the
 execution (e.g. set("infolevel",4) ).
 Displayed sequents have the form [LV],[L1]=>[RV],[R1] . The parts of the
 sequent contain the following formulas: LV  and RV : variables, L1 : v ,
 -> , <-> formulas, R1 : & , <-> formulas,

EXAMPLE

> provable((p0 -> (p1 -> p2)) & p1 -> (p0 -> p2));
  true
> provable(p0 v p2, [p0 v ~p1, p2]);
  true
> set("infolevel", 4);
  true
> provable(p0 v p1 <-> p1 v p0);
cpc::provable( A = p0 v p1 <-> p1 v p0 )
[],[]==>[],[p0 v p1 <-> p1 v p0]
(r<->)
(rv)
1.[],[p1 v p0]==>[p0, p1],[]
(lv)
1.1.[p1],[]==>[p0, p1],[]
axiom (id)
1.2.[p0],[]==>[p0, p1],[]
axiom (id)
(rv)
2.[],[p0 v p1]==>[p1, p0],[]
(lv)
2.1.[p0],[]==>[p1, p0],[]
axiom (id)
2.2.[p1],[]==>[p1, p0],[]
axiom (id)
Result of cpc::provable: true.
  true

SEE ALSO
 bddsat, consistent, satisfiable
```

Figure 6.11: MacLWB: ASCII based help system

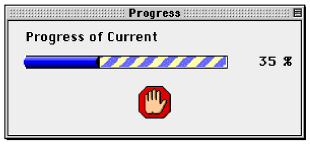


Figure 6.12: MacLWB: progress indicator

6.3 Implementation

This section contains some information about the implementations of the features presented above. The information below only shows a rough overview of how things were implemented, detailing important parts only. Additional information can be found in the source code.[4]

After some notation remarks and a short overview over the predefined classes of the POWER PLANT library, a first look goes to some general features that required implementation for the MacLWB. Afterwards, a short look to the main classes used to implement the user interface is made. As mentioned, the information given will not go into much detail.

6.3.1 Notation

This chapter uses several figures displaying the structure of and relations between the classes used. The notation used in these diagrams is shown in figure 6.13.

Relations between classes are shown if there is an inheritance between two classes, one class is a member of another or a class is referenced from another one through a pointer. Furthermore, additional relations between classes are shown when they use each other in an other, important way. For example, if a class is used in an interface or through another, not listed intermediate class.[5]

The diagrams only show own classes and the first level of the system classes. No additional details are shown for the system classes, for example the ones provided by POWER PLANT. For more details on these classes see [24]. Thus a lot of classes used in the creation of the user interface are not directly shown on the following figures.

6.3.2 POWER PLANT Predefined Classes

As mentioned in 3.1.2, POWER PLANT offers a lot of features to make the implementation of graphical user interfaces easier. To achieve this, POWER PLANT offers many predefined

[4] in the directories `interfaces/PowerPlant` and `systems/PowerPC`.

[5] this is especially the case when a class is used in a window or dialog and put there in the Constructor; while the window clearly uses the class, it is not necessary visible in the code.

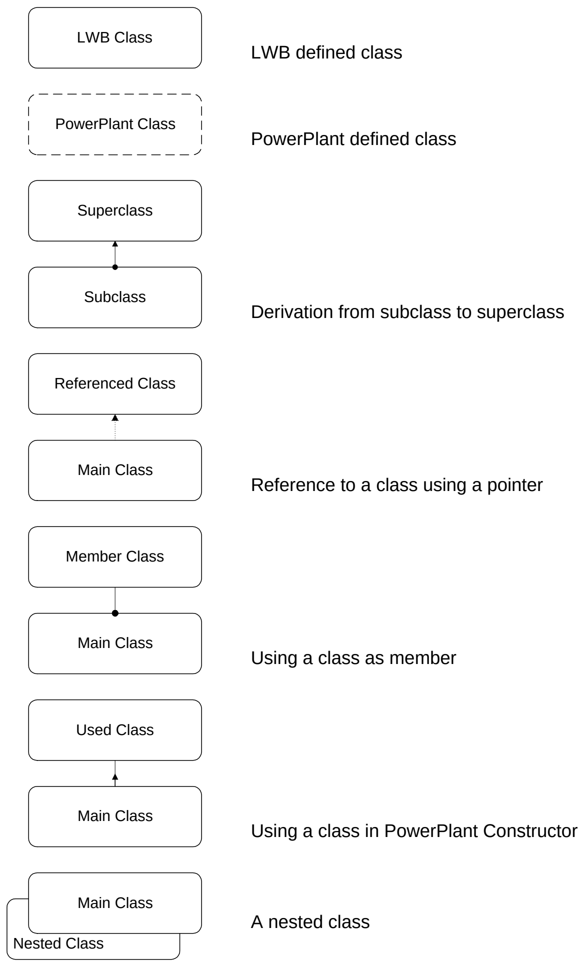


Figure 6.13: Class Diagram Notation

classes that can be used. These classes can either be used directly or—if they lack some desired functionality—they can be derived and extended with additional code.

The following gives a short, incomplete[6] summary of the classes used:

Display Classes	
`LPane`	this is the base class for all displayable GUI elements used in POWER PLANT.
`LView`	this is a class that can hold several panes; it allows scrolling of a pane inside the view, to display a pane that is larger than the window, for example.
`LCaption`	this is a POWER PLANT class used to display simple static texts, i.e. captions.
`LGACaption`	like `LCaption` this class shows some static text, this time with additional background color handling; all classes starting with `LGA` feature background handling.
`LGACheckbox`	this shows a check box in the user interface.
`LGARadioButton`	this is a display element for a radio button.
`LGAEditField`	with this class, a simple editable text field is displayed.
`LWindow`	this is the class for a basic window, supporting all the standard behavior of Macintosh windows.
`LDialogBox`	this is a derivation of the `LWindow` class, additionally supporting button handling.
`LGADialogBox`	this class is similar to the `LDialogBox` class but offers a more modern display style, for example background handling.
Utility Classes	
`LCommander`	this class is not directly visible in the user interface, but instead works behind the scenes to provide the mechanism used for handling menu and keyboard commands.
`LHierarchyTable`	this class is responsible for the management of the cells of a hierarchical table; in the case of the LWB this means the management of the regions. Unfortunately, this class does not support the automatic placement of variable sized cells.
`LPreferencesFile`	this is a special class for storing preferences values in a file; it mainly supports localizing a preferences file and some basic reading and writing methods.

[6] incomplete, because we only look at the classes we will later derive from; some of these classes are derivations again, but those won't be detailed.

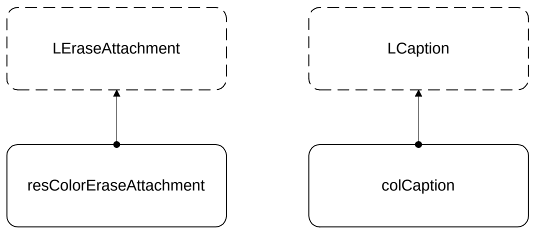


Figure 6.14: Classes for background color handling

6.3.3 Background Color

At the time the MacLWB was created, the Macintosh operating system did not yet support colored backgrounds. All elements of the user interface generally had a white background. The POWER PLANT library already supported some basic coloring of buttons and the like, but there were still some parts missing. For example, it was not possible to make an input region with a background other than white.

Because the LWB uses background colors quite heavily to distinguish input from output text, it was necessary to implement correct background handling for all the GUI elements used. For most layout elements this meant to make sure that the color was set to the preferences value when the object was created. This had do be done in addition to the normal initialization of the standard classes, because most POWER PLANT classes don't support background colors. For some of the classes this was not enough, mainly for the edit fields. Because edit fields tend to change quite a lot, they are often erased, which means they are overwritten with the background color. In the case of POWER PLANT this meant that the fields were reset back to a white background. This had to be prevented by erasing the background with the correct color at certain points of the POWER PLANT layouting process.

Along with the correct handling of the background color, it was also necessary to implement a way that allows the user to set these colors. This was done in the preferences (see below).

Most of the background color handling is implemented in the various classes for the GUI elements they represent. We briefly look at two classes here. First the `colCaption` class, because this class will not be detailed below and second the `resColorEraseAttachment`, a class which is an auxiliary helper class to enable other classes to correctly handle their background color. As shown in figure 6.14, the classes are simple derivations of standard POWER PLANT classes, adding some additional features.

`colCaption`

The `colCaption` class adds background handling to the standard POWER PLANT caption class. The class is capable of reading and writing its color values directly from the preferences. Thus, it is more easily integrated into the MacLWB preferences structure than `LGACaption`.

`resColorEraseAttachment`

POWER PLANT supports an attachment mechanism allowing the attachment of instances of certain classes of objects[7] to other objects[8]. These attachments will be called at predefined positions in the layouting and event process of POWER PLANT, for example before layouting an object or before actually handling an event. It allows the attachments to carry through some special task for different classes. Thus, it is easier and more flexible than deriving a class.

The standard POWER PLANT class `LEraseAttachment` can be attached to any drawable object to automatically erase its background before drawing it. This is enhanced by adding correct background handling with values from the preferences in `resColorEraseAttachment`. This attachment can be added to POWER PLANT classes as well as to own classes to add background handling.

6.3.4 Printing

Printing of individual regions or a complete session is not supported in the X Windows or ASCII versions of the LWB, mainly because printing on Unix systems requires the generation of Postscript files, which is quite complicated. The POWER PLANT library, on the other hand, quite nicely supports printing by allowing the creation of a virtual display containing the information to be printed and then automatically convert this to Postscript, using the Macintosh operating system.

Although POWER PLANT helps a lot when doing this, there are still several important things that had to be done. First, the virtual display had to be created, a step which is not all that hard, because this display is static and can't be changed by the user. What had to be done though, is the correct computation of page breaks. If possible, page breaks should not occur in the middle of a region, but with possibly very large regions spanning more than one page, this cannot be avoided all the time. In such a case, the page break should not occur in the middle of a line, of course. Furthermore, page counters, footers and headers have to be computed as well.

This required several changes and enhancements to existing classes to seamlessly support printing of regions. Mainly the classes `editRegion` and `regionTableCell` had to implement support for printing (see below for more details).

6.3.5 Menu and Keyboard Commands

Whenever a user presses a key or selects a menu entry, he expects some sort of reaction from the program. Sometimes, this is as simple as just displaying the key pressed, but other commands

[7] derivations of `LAttachment`

[8] derivations of `LAttachable`

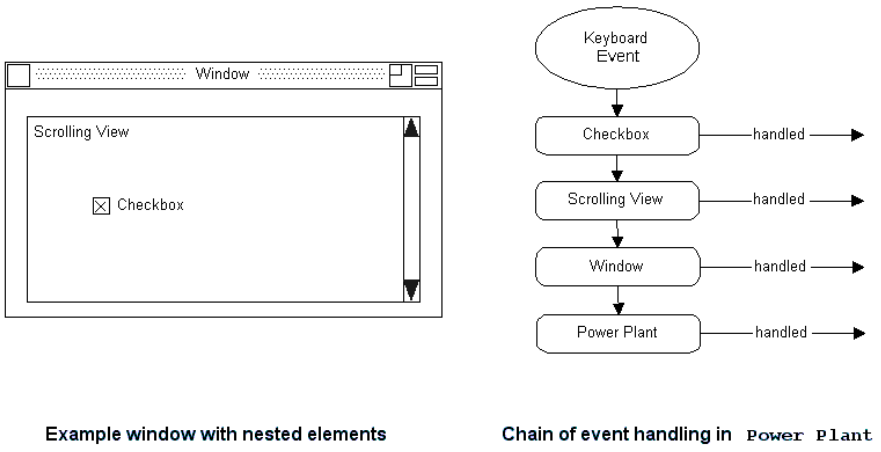


Figure 6.15: Handling of keyboard events

require more than that. In 6.2.2 is an overview of the keyboard sequences and menu command that are supported by the MacLWB.

POWER PLANT uses a special mechanism to deal with keyboard and menu commands. These commands are translated into events that are sent to the currently active interface element. This element can either directly handle the event or pass it up to its enclosing object.

Figure 6.15 shows an example of a simple window with a check box enclosed in a scrolling view. It also shows the chain used to handle the keyboard event issued from the check box.[9] This mechanism allows treatment of element specific commands in the element itself but may relegate treatment of more general commands to superior elements. This way, commands directly concerning the current element, can be implemented directly in the class of the element, while passing on all other commands without the need to treat them in any way. It requires a strictly hierarchical organization of all the interface elements, to make sure that these commands are handled.

The handling of these commands made no additional classes necessary. Instead, all classes that need to react to menu or keyboard commands define the appropriate handling methods which are called when necessary.

6.3.6 Font Handling

Like most Macintosh applications, the MacLWB supports different fonts for the different parts of the user interface. The fonts can be selected by the user in the preferences.

POWER PLANT itself supports fonts for all of its display elements containing text. Thus, the MacLWB only has to make sure, that all display elements get and display new font settings

[9] cf. chain of responsibility in [19].

```
and 236
v 235
->231
<->234

llplus  188
times  187

~ 194
dia   215
box 249

false 217
delta 182
mu 181
pi 230
Sigma 183
Pi 184
pi 185
Omega 189
chi 244
lambda 251
```

Table 6.1: example of a font translation file for the Konstanz font

whenever these are changed. This was solved by providing methods in each relevant class that either directly update the fonts and the display or pass it on to their content classes.

Furthermore, the LWB supports so called ’pretty’ fonts. These are fonts that contain special symbols for some or all of the logical operators used in the LWB. Unfortunately, different fonts will most certainly have the special symbols at other positions inside their font definition. To be able to support different fonts nonetheless, so called ’font translation’ files are used. These files describe the positions at which the symbols used can be found in the font. When a new font is selected, the MacLWB checks if there is a font translation file and if one is present, loads and uses it to print the defined logical operators. Table 6.1 shows an example of a font translation file.

One thing is not possible[10] with these font translation files. If the input and the output regions don’t use the same font, then copy and paste between these regions may fail. When copying some text, the characters are taken as is. Thus, when the text is used with another font, the wrong or no translation takes place. This could actually be solved translating the copied text into the standard LWB format and by translating it into the appropriate font translation when pasting. This would require interference with the Macintosh operating system, because the operating system handles all copy and paste operations.

[10] or at least not yet implemented.

6.3.7 Object Construction

Most of the interface elements used in the MacLWB are arranged and positioned with the CONSTRUCTOR tool. Thus, POWER PLANT needs a special mechanism to initialize these objects, using the values defined by the CONSTRUCTOR. As so often on the Macintosh, this is done using resources. The CONSTRUCTOR tool stores all values concerning interface elements in a special resource which is read when an object is created.

The predefined classes of POWER PLANT do this automatically and thus are automatically initialized. Own classes, on the other hand, need to read the initialization data when they are instantiated. The CONSTRUCTOR tool allows the setting of these values as well, thus the POWER PLANT concept for initializing can easily be incorporated into own classes.

Some of the objects also need so called 'on the fly' constructors. Such a constructor creates an object dynamically, without using any values from resources. This is mainly necessary for the regions, because the number of regions cannot be determined beforehand and regions have to be added and removed dynamically while the program is running.

6.3.8 Preferences

A major part of the user interface deals with the preference values. This part has to deal with all configuration values that can be adjusted by the user. They influence the graphical user interface as well as adjusting the behavior of the LWB itself.

As described in 6.2.4, the MacLWB offers two dialogs dealing with preference values. These two dialogs are implemented in their respective classes (`prefDialog` and `fontDialog`) and have to set the configuration values internally. Furthermore, they have to load the stored preferences values at startup and to store changed values when the user saves the preferences. When the user applies (or saves) preference values, these changes have to be incorporated into the user interface and the LWB kernel, updating interface elements, if necessary.

A distinction is made between the preferences values for the graphical user interface and for those setting internal states of the LWB. Setting, storing, and changing of configuration values for the GUI is done in the preferences objects itself and thus automatically handled correctly.

The internal LWB values have to be configured differently. Because the values are used for all LWB versions, they have to be stored and handled in a way compatible with all systems. Furthermore to prevent discrepancies between values of the user interface and values of the kernel, only one set of variables is stored for all versions of the LWB. Therefore, the LWB kernel had to be changed to take its configuration values from an abstract interface function instead of directly using it. This allows different versions of the LWB to implement different interface methods, specially targeted for an operating system. For the MacLWB, this means to implement an abstract interface that takes the values from the preferences, the same as for values of the user interface.

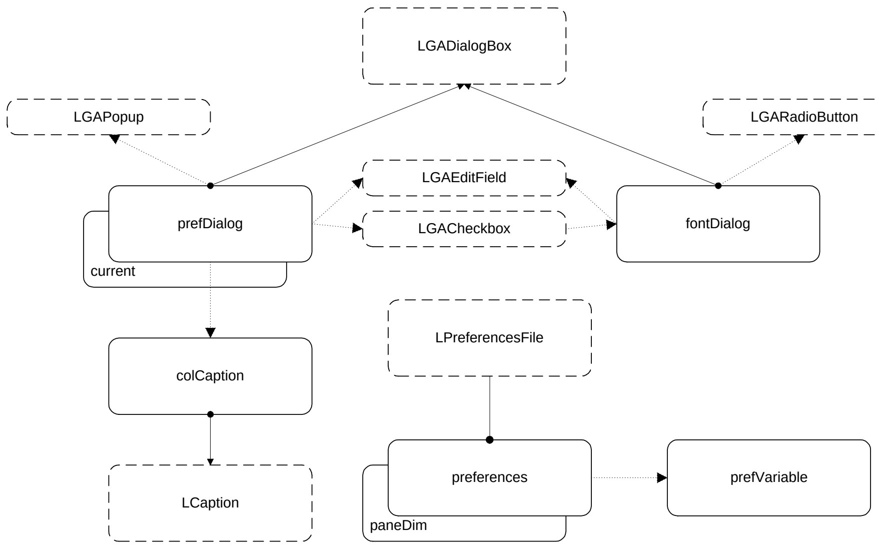


Figure 6.16: Class diagram of the preferences classes

These abstract interfaces and the dialogs were not the only thing that needed to be done when implementing the preferences. The configuration needs to be stored in and loaded from a file. On the Macintosh, special care has also to be taken to make sure that the values are stored in the proper place.[11] POWER PLANT partially supports the storing and loading of preferences values by providing a class `LPreferencesFile`. This class only supports basic file handling. Thus, functions had to be added to be able to save and load individual values as used by the MacLWB.

The classes used for handling the preference , including the dialogs, are detailed below. Figure 6.16 shows the class diagram for these classes.

`prefDialog`

This is the main dialog used for setting preferences values. This dialog displays the current settings of the LWB and of the user interface. It allows to store the values and provides means to change all configuration values. The nested `current` class is just a simple storage class used to hold the current directory for file queries using the Macintosh file dialogs.

The class mainly fulfills the following tasks:

Value Management
The `prefDialog` manages a set of values for the current settings and a set of values for the currently displayed values, to allow a user to change the values without changing the stored values. This is necessary to allow a user to cancel his changes and to revert to the last saved values.

[11] i.e. the Preferences Folder, wherever on the system it is.

Commands
The class listens to commands, mostly pressed buttons, and execute their associated actions, like

- get a color for an area's background or foreground
- get a font, using the `fontDialog` class and dialog
- revert to the last saved values
- revert to the default values
- apply the changes made to the configuration to the LWB kernel and to the user interface

Display Update
The class adjusts its display to react to changes of configuration values, be they from user manipulation in the dialog or by the execution of an LWB command. The dialog has also to be changed if a value is changed through LWB commands.
External Program
The class manages filenames and directories of the helper applications, the web browser and the Proof Wish tool. This is done by allowing the user to select an arbitrary program to be launched either for the help system or for the proof display.
Values
The `prefDialog` makes sure that preference values are read at startup. It also provides functions to load and store the preferences values.
Closing
Instead of closing the window and destroying the object, the `prefDialog` merely hides the window when it is closed. This speeds up the display and also allows to easily store the values in the class for later use.

`fontDialog`

This class is fairly simple and only needs to support the handling of the user interface. This means handling the commands issued by pressed buttons and the like and then adjusting the display as necessary. Furthermore, the class has to provide functions that allow the `prefDialog` to set the initial values to start with, as well as a means to return the values selected by the user.

`preferences`

The `preferences` class is responsible for managing all configuration values of the MacLWB. It mainly contains a list of preferences values for each type of data stored.

The class handles the following tasks:

Resource File
The class handles the Macintosh resource files used, i.e. writing and reading the actual files.

Values
As main part, the class provides methods to get individual configuration values stored. Values of the following types are possible:

- color,
- font characteristics (type, name, size, color),
- integer,
- string,
- dimension (with and height of windows).

The values are automatically read from disk the first time they are used and are written back if they were changed when the instance is destroyed. Furthermore, the configuration values can be reverted to the last saved or set back to the default values.
Dimensions
The class can adjust the dimension of a window to a dimension stored in the configuration. A dimension of a window can also be stored in the corresponding preferences value.
Files
The class allows to save the values of the configuration variables to a user defined file or to load the configuration values from such a file.

The nested class `paneDim` is used to provide a place to store the dimensions of a pane, including windows. It does not provide any methods other than a constructor and is solely used to store values.

prefVariable

While the `preferences` class manages the whole set of preferences variable, this class is responsible for the management of a single configuration value. The `preferences` class uses an instance of this class for each configuration value to be stored.

The class provides the following features:

List Management
The class provides simple list management by providing a pointer to the next value. This is used to manage multiple values in `preferences`.
States
The class handles three different states of a value in the resource file: A default value, the last saved value and the currently set value.
Resources
The class automatically releases the resources when the variable is destroyed. Furthermore, it can save a resource to the resource file and tell the rest of the list to do the same. It can also revert its value to the saved or default state of the variable. This is also propagated further down the list.

Files
The class can not only save its value to a resource, but can also save and load the value and the rest of the list to or from a normal text file.

6.3.9 Regions

The regions are the most important part of the interface of the MacLWB. Here, commands are entered by the user and the results are given back. This is also the part of the user interface which changes most. Contrary to most other parts of the user interface, the regions are not static but instead will often change in size and position. Furthermore, the user can add additional regions or remove existing ones.

Unfortunately, the POWER PLANT library only rudimentary supports fields in which text can be entered. While these classes support inserting and deleting of text, word wrapping etc., they totally lack support for dynamically sized regions. The POWER PLANT classes support the resizing of a single element of the user interface, but they do not automatically allow the repositioning of adjoining elements. But because the input and output regions of the LWB directly follow each other, a size change of one region has to change the positions of the following regions. Adding even more complications, POWER PLANT internally uses no less than four different coordinate systems to store positioning and dimension values, . Furthermore, methods and user elements had to be added to support the expansion and collapsing of regions.

Several classes had to be implemented to handle a single region and to put multiple regions into a list capable of correctly displaying all its regions, even if the size of one of its regions changes. This has to be done by computing and setting the positions for the region whenever one of the regions changes its size. These positioning adjustments had to be incorporated into the POWER PLANT layouting process, without disturbing existing drawing procedures and by making sure that the positions are not changed again by POWER PLANT. This includes the general positioning of the interface elements as well as cursor positioning and scrolling.

The layouting process of the Macintosh operating system is quite inefficient, because many drawing operations are done multiple times, to make sure the resulting display is correct. The additional level of the POWER PLANT layouting makes it even slower by adding even more repetitions. Additionally adding positioning and resizing operations for the regions, resulted in a display that was much too slow. Considering that the LWB tends to output quite a lot of text, this situation had to be greatly improved.

In the first version of the implementation, the display of the regions took much more time than it took the LWB to compute them. Therefore, the speed of the layouting process was improved. This was done first by ensuring that display changes were only done when really necessary, thus preventing multiple drawings. In a next step, text added to a region was not shown straight away. Instead, the display was updated only after a certain time passed. Thus, instead of updating and redrawing the display each time a short text is added to a region, the display is updated after some time only, incorporating all texts added since the last update. Much less updates had to be

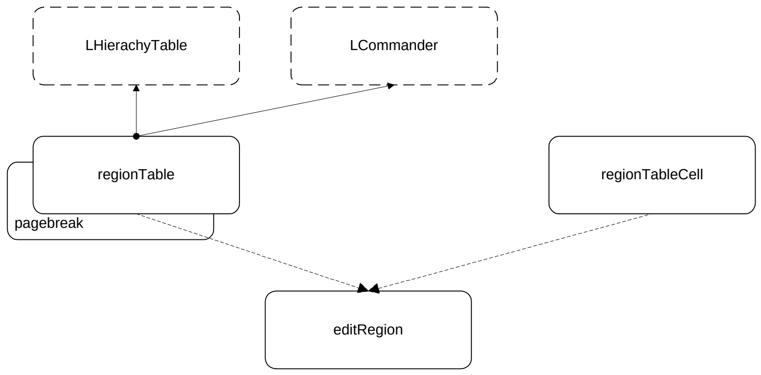


Figure 6.17: Region Classes

done that way, resulting in a much faster output with only a very short delay of showing results. As a last step, the layouting process had to be adjusted to remove flickering caused by many successive redraws. While the last step did in no way increase the speed of drawing, it made it visually faster. Now a user can no more distinguish between time used for displaying and time used for computing the results.

The implementation of all region specific parts is done in the classes `editRegion`, `regionTable` and `regionTableCell`. The first class actually handles the region itself, while the last two classes are used for positioning multiple regions in a window. The class diagrams for these three classes is shown in figure 6.17

editRegion

The `editRegion` class implements the following features:

Object Creation
The class is capable of creating objects not only using the Constructor resources but by directly constructing an object on the fly. This is necessary because a user can add an arbitrary number of regions while the MacLWB is running. Thus at startup, only one set of input/output regions is created using the Constructor method and all other regions are constructed and added to the window on the fly.
Background Color Handling
As mentioned, the `editRegion` has to correctly handle its background color, even when text is changed in the input field.
Command
The keyboard commands dealing with a single region have to be handled by this class. This includes name completion, command processing, cursor positioning and selection.
Size Adjustments
The class has to automatically adjust its size to match the amount of text entered. This is quite

time consuming, because not only the size of the current region needs to be computed, but it may become necessary to adjust the positions of the following regions as well.

Scrolling

Regions need to be able to be scrolled, either because a single region is larger than the window it is in or because multiple regions exceed the size of their window. A region has to be able to scroll itself to an absolute position, allowing the superior interface elements to scroll the regions according to the wishes of the user.

Execute Insert, Delete and Print Commands

These three commands have to be handled by this class, although their keyboard sequences are handled by the superior classes. The superior classes just delegate the commands to each individual region affected.

Text Adding

The `editRegion` class provides a method to allow other parts of the MacLWB to add text to a region. This method takes care of the size limits for edit fields on the macintosh[12] and initiates the updates of the display in specific intervals.

Reset

A region needs a way to be reset, i.e. to clear all its contents. This can either be as a reaction to a user command or through some other part of the LWB.

Information

The class needs to provide some methods returning information about the current state of the region. This includes the regions height and position as well as the text currently shown.

Expand and Collapse

The region needs to be able to expand or collapse itself. For output regions this means to hide itself completely, while input and text regions only reduce itself to display their first line. Again, this can either be the reaction to a user command or a direct call of the method by some other part of the program.

Cursor Positioning

The class provides a method to set the cursor to a specified position in the text. This is mainly used to allow cursor navigation between regions, for example if the cursor is moved from the last line of a region to the first line of the following region.

Bracket Matching

To make input safer and easier for the user, closing brackets entered are automatically matched with their opening brackets, visibly showing which two brackets belong together. If a bracket is entered without a matching opening respectively closing bracket, then a acoustic message is played.

Selection Handling

A region has to support the various ways of selecting text. The user can select text either with the mouse, by cursor keys or menu commands.

Positioning

To make dynamic sizing and positioning of regions possible, they must provide a means of positioning or moving them. This way, an enclosing interface element can move a region to a

[12] no more than 32 KBytes of text may be present in an edit field.

new position according to a size change of another region.

Printing

A region has to be able to print its contents, including handling of page headers, footers and correct page counting.

`regionTable`

The `regionTable` is responsible for handling the list of regions in the main window. It has to manage all the positions and sizes of the regions and to readjust them, if necessary. This required the following parts to be implemented in the class:

Drop Flag

The `regionTable` is responsible for drawing the drop flag, the little triangle left of each input region. Because the drop flag is not directly a part of the region itself, its management is done in the region's enclosing class, i.e. the `regionTable`.

Expand and Collapse

Like regions, the `regionTable` supports collapsing and expanding. But this time, this means to collapse or expand all regions at once, not just a single region. Thus, The command has to be redirected to each individual region.

Insertions and Removals

The table allows the insertion and removal of input/output region pairs and of text regions, as directed by user commands or other program parts. Again, the positions of affected regions have to be adjusted accordingly.

Positions

The table takes care of all positions of its regions. This includes handling when a region changes size or when a new region is inserted into the table. In these cases, the correct position changes are communicated to the affected regions.

Scrolling

The whole table supports scrolling, allowing more and larger regions to be used than actually fit in the window. The table supports automatic scrolling when text is entered or the cursor position is changed, as well as user initiated scrolling when the scroll bar is hit or page up or page down keys are pressed. The table determines which parts of which regions have to be displayed and sets the values for drawing accordingly.

Command Execution

All the keyboard and menu commands that are not directly handled by a region or that may be called without a current region, are handled in the `regionTable`. This includes the commands to resize a region, add text to a region, scroll the table or a region, delete a region, move the cursor between regions, and the collapsing and expansion of multiple regions.

Font Update

An update of fonts originating from a superior class is routed to all the individual regions in the table.

Printing
The table computes the page breaks that are necessary when printing the whole session, i.e. all current regions.
Reevaluation
The `regionTable` supports the reevaluation of all input regions currently stored. This means that all input commands are reexecuted, starting with the first region.
Information The table only supports access to the whole text of all presently stored regions, but not to the text of individual regions.

`regionTableCell`

This is an auxiliary class used by the `regionTable` class to store the information of each table cell. This class stores a pointer to the actual `editRegion` along with the regions type. It is only a wrapper for the `editRegion` class. The main purpose for this class lies in the structure of the POWER PLANT provided `LHieararchyTable` class. This class uses small, simple objects to store the contents of each table cell. Because these objects are copied when adding them to the table or when moving them around, it's more efficient to use a wrapper class instead of directly using the whole `editRegion`.

The class does not fulfill any other tasks and does not have any functions beside its constructor and destructor.

6.3.10 Progress Indicator

Another, on the Macintosh especially important part to be implemented was the progress indicator. It displays the amount of a computation already done. On the Macintosh it also allows to interrupt the current computation.

The actual computation of the percentage values is part of the LWB algorithms and was reused without problems. This information has to be displayed, though. This required to add some code to display values, but also required changes in existing code. Earlier versions of the LWB directly wrote the percentage information to the screen.[13] Again, the main part of the LWB and the user interface were decoupled using an abstract interface (cf. 5.2.2). Thus, when the percentage value changes, an interface method is called. This method, depending on version of the LWB, does the things necessary to update the display showing the value. For the MacLWB this means to adjust the progress bar.

The user interface of the progress indicator must fulfill another important task. It must provide the means to interrupt the current computation. This is done by providing a button which initiates the interrupt. How the computation is actually interrupted is described in the next section.

[13] more precisely to the info window.

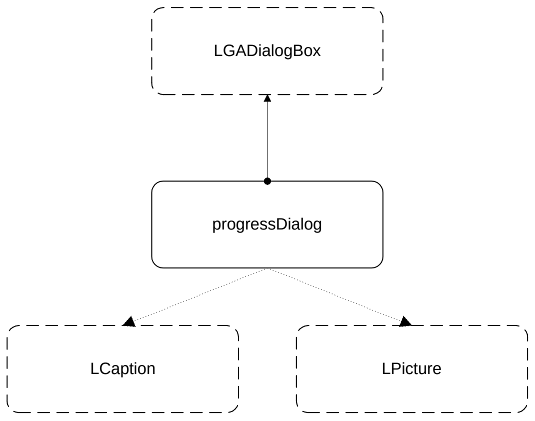


Figure 6.18: Class diagram of the progress dialog class

The implementation of the progress indicator—excluding interrupt handling—is done in the class `progressDialog`. The class diagram for this class is shown in figure 6.18.

`progressDialog`

This class implements all the layout specific tasks concerning the progress indicator. As already mentioned, the actual computation of the percentage value is done in the LWB algorithms and the interrupt handling is done in the system specific part (see below).

The implementation has to respond to the interrupt request either from the button or from a keyboard or menu command. As with other windows, this window is also merely hidden from view when it is closed, to allow faster display. Of course, the class has to provide a method allowing the LWB kernel to adjust the value displayed.

6.3.11 Interrupts

Interrupts are used in the LWB to stop a lengthy computation right in the middle of its execution. This can be used as a means to stop a computation that's not worth waiting for or that simply takes more time than is acceptable. The implementation of interrupts faces some problems. The main problem when interrupting a program in the middle of its execution is that its current state is unknown. Such an interrupt may easily result in inconsistent data, locking of resources (files, devices, memory), and general loss of memory. Furthermore, somehow the program has to continue, if possible with as little loss of data as possible and by returning into a stable state.

Earlier versions of the LWB used the interrupt handling mechanisms provided by Unix to interrupt program execution. This mechanism allows to store the state of the machine (program counter, registers, stack pointer) in a special data structure. Later in the program execution, as a reaction to the interrupt, this state can be reloaded, forcing the program to continue right after storing the state, just as nothing had happened. This approach has the advantage to be easy to implement. It only requires to add an initializing statement to save the state and an interrupt handling routine which reloads and sets the machine back to a previous state. Thus, most part of a program does not have to be changed. It has some serious drawbacks, though. First and foremost, the Macintosh does not have a similar interrupt handling mechanism. While it is possible to interrupt a program on the Macintosh, this does completely stop the program. It is not possible to interrupt a program and give it some way to react to the interrupt. Furthermore, the program is interrupted at an arbitrary position in the program code. While the restoring of the machine state adjusts the stack and the program counter to resume normal program execution, it does not otherwise handle memory considerations. All memory that was not allocated on the stack cannot be freed again and is lost. Furthermore, any cleanups that should have been done when freeing such memory is not done as well. Worse, data may have been partially initialized or modified and can now contain inconsistent data. Using such data later will probably create serious problems.[14]

To take care of both of these disadvantages, a new, different interrupt handling mechanisms was implemented, on Unix and especially on the Macintosh. Instead of allowing an interrupt at arbitrary program position, an interrupt is only handled at certain positions. To achieve this, each algorithm calls, at certain, safe positions, a periodical function. This function checks if an interrupt occurred and if necessary handles it.

Instead of restoring a previous machine state to handle an interrupt to return to a stable program state before the computation, an interrupt is handled by using the exception handling mechanism of C++. This mechanism allows a chain of positions to be defined while a program is running. This positions can, if necessary, be resolved backwards using the exception mechanism. As a result, the program can resume at a desired, previous position.

This mechanism replaces direct restoring of machine states in a portable way, supported by the C++ language. Furthermore, each affected function may include special exception handling routines that allow to clean up resources, free memory and to make sure all data is consistent. This mechanism can theoretically clean up all resources, including memory. This requires special clean up routines each time memory is allocated. This means to make lots of modifications to an existing program. But even if no additional routines are added, this solution is not worse than the previous one and with little effort, the most important memory losses may be prevented. Furthermore, future algorithms could provide the necessary exception handling routines from the start, allowing a complete clean up of their resources.

As a result, existing algorithms only need to define the positions in their execution, where an interrupt can be allowed. This is done by calling a special function at these positions. This function checks for interrupts. If each algorithm makes sure that this function is called periodically, for example at least once per second, the program can respond with little or now delay to an interrupt request by the user.

[14] of course, algorithms should always correctly clean up all memory previously allocated when terminating normally.

For the Macintosh, this solution has several benefits. First, it allows the implementation of interrupt handling at all. Second, periodically calling a function also allows to periodically give the user interface time to update its display and to react to user input while a computation is under way. Without this, the graphical user interface is blocked and does not react in any way to the user. The user is only able to move the mouse pointer, but windows moved or resized are not redrawn until a computation is finished. With this new approach, the user is even able to switch to another application while the MacLWB still continues to compute. Furthermore, the user can actually see that the computation is still done and that the computer is not crashed. This includes the updating of the progress indicator, of course. A third benefit is the timing features that are now possible (see next section).

The following two classes implement interrupt handling. As all system specific parts, both use an abstract interface (cf. 5.2.2) to decouple the implementation of the methods from their callers.

`interrupt`

This is the main class responsible for interrupt handling. It handles interrupts by throwing an interrupt exception and disabling further interrupts, to prevent additional interrupts while one is being handled. It also provides a method to check if an interrupt is pending or if one is currently handled. Furthermore, methods allow to enable, disable and initialize interrupt handling.

`periodical`

The `periodical` class is not directly involved in the interrupt handling process, but instead provides a means for the LWB algorithms to periodically do certain things, including checking for pending interrupts. The class contains a single method, which is periodically called by each algorithm of the LWB. On the Macintosh, this method makes a limit check to find out if a time limit has passed (see next section), checks for low stack space to warn the user and processes Macintosh events. The last allows interrupt handling and gives the user interface time to refresh its display while a computation is done.

6.3.12 Timing

The LWB provides commands to determine the time used by a computation. Adding the periodical and interrupt tasks previously mentioned, allowed to add commands to set up a time limit for a computation (`limitstart()` and `limitstop()`).

`systime`

Part of the timing process, i.e. the `periodical` and `interrupt` classes were already mentioned in the previous section. The rest of the timing specific code is encapsulated in the class

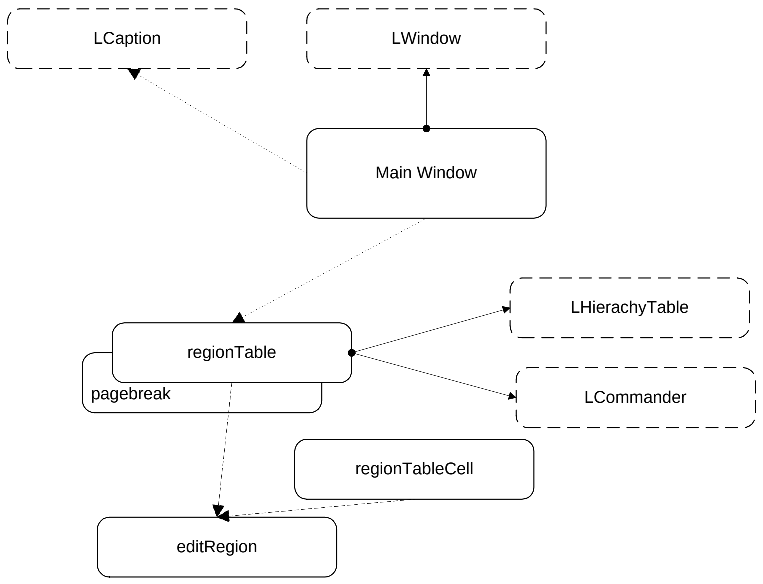


Figure 6.19: Class diagram of the main window

`systime`.[15] This class provides an abstract interface with methods to get the current number of clock ticks and the number of clock ticks per second. Furthermore, it is possible to directly get the amount of 1/100 s passed so far, thus allowing to determine how much time has passed.

6.3.13 Main Window

As mentioned in 6.2.1, the main window deals with the user input and with displaying results.

`mainWindow`

The main window is implemented through the class `mainWindow`. Its class structure is shown in figure 6.19. The classes `regionTable`, `editRegion` and `regionTableCell` are described in 6.3.9.

The `mainWindow` class is derived from the POWER PLANT class `LWindow`, the POWER PLANT class for general windows. This base class does most of the layouting and basic event handling for the window. The `mainWindow` class has to fulfill the following additional tasks:

Quit
When the window is closed, after asking the user for confirmation, it has to quit the whole MacLWB.

[15] the name `time` is already used on Unix systems.

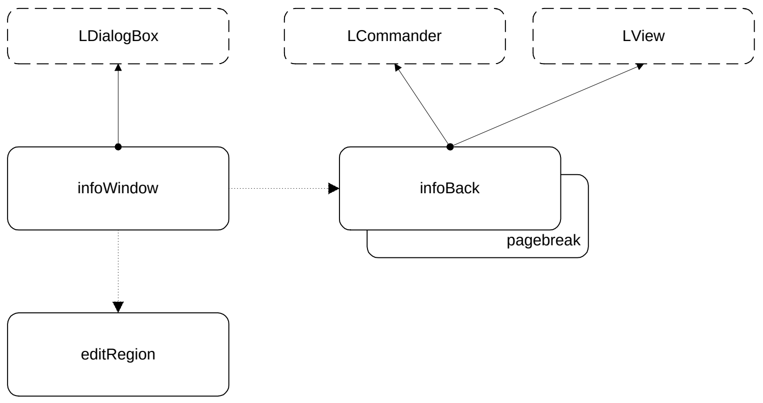


Figure 6.20: Class diagram of the info window

Commands
Most of the keyboard and menu commands are handled by this class or must be relegated to the appropriate interface elements.
Preferences
Preferences values for dimensions, fore- and background colors are read—and used–when first opening the window.
Regions
Methods are available to write to the current output region, to update fonts and to show an error. Furthermore, it is possible to insert input/output and text regions and a method returns the text of all current regions.
Modules
The class handles the display of which modules are loaded, i.e. shows when modules are added, removed or changed in sequence.
Memory
Finally, the class shows with the memory display how much memory is available for computation.

6.3.14 Info Window

The info window is responsible for displaying status and debugging information of the LWB algorithms.

`infoWindow`

The Info Window is implemented in the class `infoWindow`. Its class diagram is shown in figure 6.20. This section will deal with the `infoWindow` class and the `infoBack` class. The class `editRegion` was detailed in 6.3.9.

The `infoWindow` class is not derived from the POWER PLANT `LWindow` class but instead from the `LDialogBox` class. This is because the info window also has buttons which need to be handled. This can be done easier using the `LDialogBox` class, which in turn is itself a derivation of the `LWindow` class.

The following tasks are implemented in the `infoWindow` class:

Preferences
Depending on the preferences values, the info window is shown or hidden at startup. Furthermore, the size, position, font and color of the window is adjusted to the values last stored.

Quit
As the other windows, the info window is merely hidden when it is closed by the user instead of really destroying the class. This way, the text in the window is saved and additional text can be added as well, even if it is not shown.

Commands
The class listens to the events issued by the reset and print buttons and execute the appropriate actions, i.e. resetting the window contents or printing the whole text.

Update
The class provides methods to update the font used, in case it is changed by the user, and also allows to refresh the text display.

`infoBack`

The `infoBack` occupies most part of the info window. It is responsible for scrolling the text display. It also manages resizing and cursor movement and provides the means to display the correct background color in the window. Finally, it is responsible for printing, mainly for determining the number of pages and the actual page breaks.

6.4 Conclusion

The information presented in this chapter only contains an overview of the implementations that were done for the MacLWB and for the LWB in general. Further information on the implementation can be found in the documentation of the source code itself. For that reason, the source code, along with executable versions of the LWB for the Macintosh, Linux and Solaris are present on the attached CD-ROM. The CD also contains the complete documentation as it is available on the world wide web.

Part II

Logic of Likelihood

Chapter 7

Introduction

The logic of likelihood LL was introduced in 1987 by Halpern and Rabin in [22]. They motivate their logic mainly with an example of a protocol verification of data transfer. The logic can be used for other things as well, though and it has some interesting relationships other modal logics and even with logics dealing with common knowledge.

Most of this part will deal solely with a subset of the logic of likelihood, called LL^-. While one important operator of the logic is not present (L^*) in this subset, it is still strong enough for the examples shown in the paper. The subset of the logic can be implemented much easier and more efficiently than the full logic.

The next chapter gives the semantics and some simple examples for the logic, while later chapters deal more closely with proof search in the logic. Chapter 9 introduces a Hilbert calculus for LL^-, which will be used to obtain the Tait calculus of chapter 10. The Tait calculus will be used as a basis for the double-sided sequent calculus of chapter 11. The remaining chapters will deal with an extensive example of the logic that was done using the Logics Workbench, followed by some remarks about the implementation of the automatic theorem prover for LL^- in the LWB.

7.1 Motivation

A lot of logics are used to formalize real world problems to be able to better understand and handle them. In this context, a set of statements is translated into the notation of the logic. The logic then helps to find out if—from this set of statements—it is possible to deduce certain facts, i.e. other statements.

While it is sometimes necessary to use a first order logic to formalize real world problems, a lot of problems can be dealt with in the context of propositional logics. This often results in a big number of propositional variables that have to be used. But, because propositional logics are often much simpler to deal with than first order logics, this step is only a notational but not always a computational disadvantage.

One category of interesting real world problems that is to deal with, is the decision making process. It is often interesting to know, given certain facts, which decision can be deduced from them. Because most real world facts are not based on certainty, likelihood is an important component to formalize and make deductions for them.

One way to deal with likelihood is to use probability theory. Unfortunately there are several drawbacks to this solution. Most importantly, it may not always be possible to attach probabilistic values to the events and facts in question. For example, in a lot of medical situations the expense and inconvenience to the patient from obtaining probabilistic values rarely justify the conclusions that can be drawn from them. Furthermore, even if the exact probabilistic numbers are available, people are often quite uncomfortable using them. Although people are ready to state that some fact is more likely than another, they are rarely willing to give exact numerical probabilities for them.

Lastly, a lot of expert systems have shown that their results do not change for small ($< 30\%$) perturbations in the numerical probabilities used. This observation suggests an approach to likelihood may be preferable that only uses a qualitative, non numerical notion of likelihood. The logic of likelihood LL by Halpern and Rabin uses this approach. It takes normal classical propositional logic and adds a modal operator L. This operator will be used to express that a formula is likely. Thus, the formula $\mathsf{L}p$ can be roughly translated to "p is reasonably likely to be a consistent hypothesis". Clearly, it is the user of the logic who decides what confidence should be given to the operator L. As we will see, it is also possible that at the same time $\mathsf{L}p$ and $\mathsf{L}\neg p$ hold.

Although the logic wants to avoid the problems for probabilistic approaches mentioned above, it is still possible to express various degrees of likelihood. This is done by nesting multiple L operators, i.e. $\mathsf{LL}p$ could be translated as "p is somewhat likely to be a consistent hypothesis". Adding even more L, would make the statement less and less likely. Furthermore, given the fact that p_2 is reasonably likely given p_1 (formal $p_1 \rightarrow \mathsf{L}p_2$), and that p_3 is reasonably likely given p_2 (formal $p_2 \rightarrow \mathsf{L}p_3$), it can be deduced that p_3 is somewhat likely given p_1 (formal $p_1 \rightarrow \mathsf{LL}p_3$). This means that likelihood behaves as expected.

In addition, the logic of likelihood contains an operator G is used to denote necessity. This allows formulas to express facts that must hold in general, without making use of likelihood.

If we compare the logic of likelihood with one of the standard modal logics, like K or S_4, we can relate the operator L to the $\Diamond$ operator of the modal logic and G to its $\Box$ operator. Although both logics are similar in many ways, it has to be noted that there are subtle, but important differences. While the $\Diamond$ operator in standard modal logic can be defined as $\neg\Box\neg$ this is not the case for the logic of likelihood. While we still have $\mathsf{G}p \rightarrow \neg\mathsf{L}\neg p$, the direction backwards is not necessarily true. We will see a more detailed example at the end of the next chapter after the semantics of the logic were introduced.

Chapter 8

Semantics

This chapter defines the semantics of the logic of likelihood LL. It concentrates on the subset LL$^-$ from [22], that does not contain the L^* operator for the transitive closure of L.

8.1 Syntax

Before we start describing how to express statements and facts in the logic of likelihood, we have to look at the syntactical structure of the logic. The following definitions take care of the basics of the logic.

Definition 1 ***Language of LL$^-$***

The language of LL$^-$ *consists of*

- *countably many propositional variables* $p_0, p_1, p_2, \ldots$,
- *the operators* $\neg$ *and* $\wedge$,
- $($ *and* $)$ *to group formulas,*
- *the modal operators* G *and* L.

The complete logic of likelihood LL has an additional operator L^*, which will be left out for this and the next chapters. Later, several abbreviations for commonly used formulas and operators, like disjunction or the dual operators to G and L, will be introduced. They are left out in the definition above to simplify the following definitions and proofs.

Definition 2 ***Formula of LL$^-$***

A formula *of LL$^-$ is inductively defined as follows*

- *each propositional variable* $p_0, p_1, \ldots$ *is a formula,*
- *if* A *is a formula, then* $\neg A$ *is a formula as well,*
- *if* A *and* B *are formulas, then* $(A \wedge B)$ *is also a formula,*

- *if A is a formula, then $\mathsf{G}A$ and $\mathsf{L}A$, are formulas.*

The set of all formulas of LL$^-$ is called $\mathrm{Fml}_{\mathrm{LL}^-}$.

The parentheses in the previous definition are omitted if they are clear from context, i.e. we write $A \wedge \neg B$ instead of $(A \wedge \neg B)$. Furthermore $\wedge$ is left associative. This allows to leave out even more parentheses, i.e. instead of writing $((A \wedge B) \wedge C) \wedge D$ we can write $A \wedge B \wedge C \wedge D$.

Definition 3 ***Theory of LL$^-$***
A theory T is a finite set of formulas of LL$^-$, i.e. $T \subset \mathrm{Fml}_{\mathrm{LL}^-}$ and T finite.

Definition 4 ***Formula Length***
We define the length of a formula $|\cdot|$ *inductively as:*

$$\begin{aligned}
|p_i| &= 1,\\
|\neg A| &= |A| + 1,\\
|A \wedge B| &= |A| + |B| + 1,\\
|\mathsf{G}A| &= |A| + 1,\\
|\mathsf{L}A| &= |A| + 1.
\end{aligned}$$

Using this definition, the length of a formula is exactly the same as the number of its symbols, counting each propositional variable as a single, different symbol.

To simplify the following definitions and propositions, we introduce several abbreviations. It must be noted, though, that these abbreviations are not a part of the language and thus are not treated in proofs and propositions.

Definition 5 ***Abbreviations of LL$^-$***

$$\begin{aligned}
A \vee B &:= \neg(\neg A \wedge \neg B),\\
A \rightarrow B &:= \neg A \vee B,\\
A \leftrightarrow B &:= (A \rightarrow B) \wedge (B \rightarrow A),\\
\mathsf{F}A &:= \neg\mathsf{G}\neg A,\\
\mathsf{K}A &:= \neg\mathsf{L}\neg A.
\end{aligned}$$

8.2 Notation

To make writing and reading statements easier, some notations will be introduced below. These notations in no way extend the language of LL$^-$, but allow to write down statements in a more compact, easily readable form.

The following notations will be used in this and later chapters:

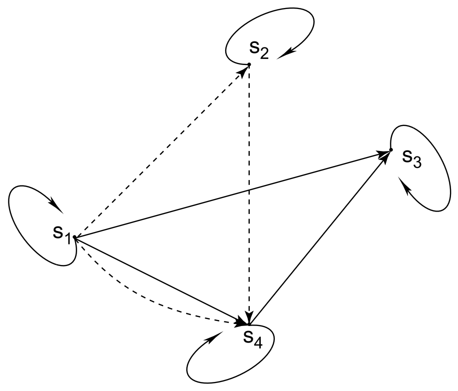


Figure 8.1: An example of a model in LL$^-$

- we use the symbols $p, q, p_0, p_1, \ldots$ for propositional variables,
- the capital letters A, B, C, and D are used to denote formulas of the logic,
- we use the letter T for theories,
- as mentioned, we omit parentheses that are clear from context
- we write $\mathsf{L}^n A$ for $\underbrace{\mathsf{L} \cdots \mathsf{L}}_{n \times \mathsf{L}} A$ and $\mathsf{K}^n A$ for $\underbrace{\mathsf{K} \cdots \mathsf{K}}_{n \times \mathsf{K}} A$,
- we use $\mathcal{M}$ for models and s or t for single states in such models.

8.2.1 Models

As usual for modal logics, the semantics for LL$^-$ is given be means of Kripke models.

Definition 6 ***Model***
An LL$^-$ model $\mathcal{M}$ *is a quadruple* $(S, \mathscr{L}, \mathscr{C}, \pi)$ *with the following properties*

- S *is a set of states,*
- $\mathscr{L} \subset S \times S$ *is a binary reflexive relation,*
- $\mathscr{C} \subset S \times S$ *is a binary relation,*
- $\pi : \{p_i : i \in \mathbb{N}\} \to 2^S$.

A comparison with a usual modal logic might help to completely understand the definition above. The set S of states corresponds to the different worlds used in modal logics. In LL$^-$ it represents different sets of hypotheses considered to be valid at each state. The relations $\mathscr{L}$ and $\mathscr{C}$ are later used to define the L and G operators, similarly to the definition of the $\Box$ operator in modal logics. From a specific state, they represent which states are likely respectively conceivable . The function π intuitively associates with each propositional variable the set of states in which the variable is assumed to be true. Thus, π is the truth function of an individual state.

Example 2 ***Simple Model***

Before we go on with additional definitions for models, we take a look at a simple example. Let

$$\begin{aligned}
S &= \{s_1, s_2, s_3, s_4\},\\
\mathscr{L} &= \{(s_1,s_1),(s_1,s_3),(s_1,s_4),(s_2,s_2),\\
&\quad (s_3,s_3),(s_4,s_3),(s_4,s_4)\},\\
\mathscr{C} &= \{(s_1,s_2),(s_1,s_4),(s_2,s_4)\},\\
\pi(p_1) &= \{s_1, s_2\},\\
\pi(p_2) &= \{s_2, s_3, s_4\}.
\end{aligned}$$

The model has four states s_1 to s_4. The states s_1 and s_3 are connected by $\mathscr{L}$, as well as s_1 to s_4 and so on. We have a similar situation for $\mathscr{C}$. Furthermore, the propositional variable p_1 is true in the states s_1 and s_2, while p_2 is true in the states $s_2, s_3,$ and s_4. As required, the definition of $\mathscr{L}$ is reflexive. The whole model is shown in figure 8.1.

To be able to talk a little more about models of LL$^-$, some additional definitions are required.

Definition 7 ***Model Size***

The size *of a model $\mathcal{M} = (S, \mathscr{L}, \mathscr{C}, \pi)$ is $|S|$, i.e. the number of states in the set of states S.*

Because the definition of a model does no put any requirements to the set of states, models in LL$^-$ can be finite or infinite.

The following definitions helps us to speak about the connections between different states.

Definition 8 ***Successor***

A state t is a successor *of a state s if $(s,t) \in \mathscr{L} \cup \mathscr{C}$.*
*A state t is a $\mathscr{L}$-*successor *of a state s if $(s,t) \in \mathscr{L}$.*
*A state t is a $\mathscr{C}$-*successor *of a state s if $(s,t) \in \mathscr{C}$.*

Clearly, because the relation $\mathscr{L}$ has to be reflexive, every state is a successor (even a $\mathscr{L}$-successor) of itself. Furthermore, if a state is an $\mathscr{L}$- or $\mathscr{C}$-successor of another state, it's clearly a successor as well.

Using our previous example shown in figure 8.1 we see that s_4 is a successor, an $\mathscr{L}$-successor and a $\mathscr{C}$-successor of s_1. s_2 is only a $\mathscr{C}$-successor and with that of course also a successor of s_1. Last but not least, s_3 is an $\mathscr{L}$-successor of s_1 but not a $\mathscr{C}$-successor.

As will be shown below, the $\mathscr{L}$ relation will be used to define the Operator L. This means, an $\mathscr{L}$-successor describes the set of hypotheses regarded as reasonably likely given the hypotheses of the current state. On the other hand, a $\mathscr{C}$-successor describes the hypotheses that are conceivable but not necessarily likely given the hypotheses of the current state.[1]

Example 3 ***Likely and Conceivable***
As an example for the two relations we regard a state describing weather situations. If our current state has hypotheses that state that we are in the month of April and located in Switzerland, then it's likely that it will rain the next day. Thus, a state where it rains the next day might be an $\mathscr{L}$-successor. Although it is not likely that it will snow the next day, it did happen. Thus, it is surely conceivable though not likely that it will snow the next day. Thus, a state where it will snow the next day is a $\mathscr{C}$-successor of our current state.

The next definition treats relationships between states that are not direct successors but connected via multiple states.

Definition 9 ***Reachable***
A state t is reachable *(respectively* $\mathscr{L}$-reachable*) from a state s, if there is a finite sequence of states $s_0, \ldots, s_k$, where $s_0 = s, s_k = t$ and s_{i+1} is a successor (respectively an $\mathscr{L}$-successor) of s_i, for all $0 \leq i < k$.*

Again, given the fact that the $\mathscr{L}$-relation has to be reflexive, every state is reachable and $\mathscr{L}$-reachable from itself.

Coming back to our previous example for a model in LL$^-$, shown in figure 8.1, we see that s_4, s_3, and s_2 are all reachable from s_1, but only s_4 and s_3 are also $\mathscr{L}$-reachable. Furthermore, s_3 is reachable from s_2, using a path over s_4, but it's not $\mathscr{L}$-reachable. Lastly, s_1, s_2, and s_4 are all not reachable from s_3.

Now we have everything we need to define the truth value of a complex formula in a specific state. As expected, this has to use information from other states as well.

Definition 10 ***Valuation***
We extend the mapping π to a valuation $\hat{\pi} : \mathrm{Fml}_{\mathrm{LL}^-} \longrightarrow 2^S$ *as follows*

$$\begin{aligned}
\hat{\pi}(p) &= \pi(p), \\
\hat{\pi}(\neg A) &= S - \hat{\pi}(A), \\
\hat{\pi}(A \wedge B) &= \hat{\pi}(A) \cap \hat{\pi}(B), \\
\hat{\pi}(\mathsf{G}A) &= \{s \mid t \in \hat{\pi}(A) \textit{ for all } t \textit{ reachable from } s\}, \\
\hat{\pi}(\mathsf{L}A) &= \{s \mid t \in \hat{\pi}(A) \textit{ for a } t \textit{ that is an } \mathscr{L}\textit{-successor of } s\}.
\end{aligned}$$

[1] it would have been possible to interpret the relation $\mathscr{C}$ as just meaning conceivable by itself, but then a relationship between the relations $\mathscr{L}$ and $\mathscr{C}$ would have to be postulated, because everything that is likely should also be conceivable.

The definition for the value of $\mathsf{G}A$ uses the union of the relations $\mathscr{L}$ and $\mathscr{C}$. That means the G operator includes all states that are conceivable, including the likely ones. This actually defines $\mathsf{G}A$ as meaning that A has to hold in any case, i.e. generally. Thus, $\mathsf{G}A$ is similar to a transitive $\Box$-operator of modal logic (for example the $\Box$ of S_4).

On the other hand, the definition of the L operator only uses the $\mathscr{L}$-relation, as expected. It is comparable to a $\Diamond$-operator, which is not transitive and does not include all successors.

The definition of the other values are the same as for classical propositional logic. If we take a look at the abbreviations we have done, then we see that for the classical operators $\vee, \rightarrow, \leftrightarrow$ nothing extraordinary happens. The operator K says that $\mathsf{K}A$ holds if A holds in *all* $\mathscr{L}$-successors. That does not mean that it must hold in all successors, because $\mathscr{C}$-successors need not fulfill this property in order for $\mathsf{K}A$ to hold. K can thus be viewed as a sort of modal, non-transitive $\Box$-operator, with the difference that not all successors have to be used. Thus, the duality of the L and K-operators from the definition nicely corresponds with the duality of standard modal logics. The operator F says that $\mathsf{F}A$ holds if there is a reachable state where A holds. Thus, it is comparable to a transitive $\Diamond$-operator, again the dual to G, respectively the $\Box$-operator.

All in all, we see that LL^- is actually comparable to a logic with two sets of modal operators, where one set is transitive and the other is not. This will be shown in more detail in 11.9.

Definition 11 ***Models***
As usual we write $\mathcal{M}, s \models A$ instead of $s \in \hat{\pi}(A)$ and $\mathcal{M} \models A$ if $\mathcal{M}, s \models A$ for all $s \in S$.

The following two definitions are the same as for other logics, but still necessary.

Definition 12 ***Satisfiable***
A formula A is satisfiable, *if $M, s \models A$ for a model $M = (S, \mathscr{L}, \mathscr{C}, \pi)$ and a state $s \in S$.*

Definition 13 ***Valid***
A formula A is valid, *if $M, s \models A$ for all models $M = (S, \mathscr{L}, \mathscr{C}, \pi)$ and all states $s \in S$. In that case we write $\models A$.*

If we take a look at the strength of the various operators, we can order them in strictly decreasing order as follows:

$$\mathsf{G}p, \quad \ldots, \quad \mathsf{K}^n p, \quad \ldots, \quad \mathsf{K}p, \quad p, \quad \mathsf{L}p, \quad \ldots, \quad \mathsf{L}^n p, \quad \ldots, \quad \mathsf{F}p$$

This means that, for example, the formulas $\mathsf{G}p \rightarrow \mathsf{K}p$, $\mathsf{K}p \rightarrow \mathsf{L}p$, or $p \rightarrow \mathsf{L}p$ are all valid. The last one clearly makes sense, if we know that p holds, it is also at least likely that it holds. The inverse of the formulas are generally not valid.

8.3 Properties

Before we take a look at some examples, we state some important properties of LL$^-$ and its models. The proofs of these statements (or at least a proof sketch) can be found in the original paper [22].

Theorem 1 ***Finite Models***
An LL$^-$ formula A is satisfiable iff it is satisfiable in a model of size $\leq 2^{|A|}$.

Theorem 2 ***Decision Procedure***
For some $c > 0$, there is a procedure for deciding if a formula A is satisfiable (respectively valid), which runs in deterministic time $O(2^{c|A|})$.

8.4 Examples

In this chapter we take a look at two examples which will show how LL$^-$ can be used to express statements including likelihood. Both examples originally are from [22], but are presented here in a slightly different form.

8.4.1 Medical

Example 4 ***Simple Medical Deductions***
As a first example, we take a look at medical patients. Each patients may show several symptoms, has a medical history and may have diseases. For each possible symptom, history data and disease we define a propositional variable. Each variable states if a patient has the corresponding symptom, history data, or disease. For example, we can use the following set of propositional variables:

Symptoms:

p_{Y}	the patient has a yellow complexion,
p_{W}	the patient has a white tongue.

History:

p_{DR}	the patient is a heavy drinker,
p_{HR}	the patient has heart troubles.

Disease:

p_{HP}	the patient has hepatitis,
p_{D}	the patient will die,
p_{T}	the patient has a tumor.

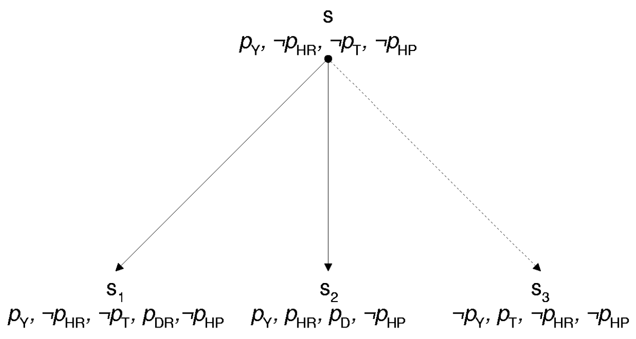

s
$p_Y, \neg p_{HR}, \neg p_T, \neg p_{HP}$
s_1
$p_Y, \neg p_{HR}, \neg p_T, p_{DR}, \neg p_{HP}$
s_2
$p_Y, p_{HR}, p_D, \neg p_{HP}$
s_3
$\neg p_Y, p_T, \neg p_{HR}, \neg p_{HP}$

Figure 8.2: Medical example: some hypotheses

Of course, this set is far from complete for a real medical application, but it is enough to show some basic properties and applications of LL^-.

A medical doctor can use the language of LL^- and the propositional variables defined above to formalize a patient's symptoms and medical history. He can use the L-operator to express his uncertainty, not only in the disease he wants to deduce, but also in the symptoms and the patient's medical history. Thus, while a doctor may not know for a fact that a patient has a drinking problem, he may still have suspicions. Thus, instead of directly taking p_{DR} as a hypothesis he can only take $\mathsf{L}p_{DR}$. Thus, he expresses that it is likely that the patient has a drinking problem, but it's not necessary so.

It has to be noted that the logic of likelihood is not a temporal logic. Thus, the successors defined for the states in a model are not temporal successors in the sense that after the current state the successor state will follow. Instead, the successors represent the sets of hypothesis that are likely respective conceivable if the current state is accepted. That means, each state represents a consistent and complete set of hypotheses which are taken to be true for now. Of course, in practice only the 'relevant' formulas will be treated in each of these sets.

From a specific set of states we can create other sets of hypotheses either as being likely or conceivable (or nothing at all) from our current set of hypotheses. This is the modeled in an LL^- model using the previously defined relations $\mathscr{L}$ and $\mathscr{C}$.

For example, we start in state s with the following hypotheses for a specific patient. The doctor observes that the patient has a yellow complexion, thus we put p_Y into our set of hypotheses. Furthermore, the medical record of the patient does not show that he has heart troubles, thus we add $\neg p_{HR}$. Then, after a close examination, the doctor is sure that the patient does not a have a tumor, thus we add $\neg p_T$. For the rest of the propositional variables we know nothing special, thus we add their negation to our working hypotheses. This gives us the state as depicted in figure 8.2.[2] We can now define hypotheses that are either $\mathscr{L}$-successors or $\mathscr{C}$-successor of our

[2] all additional propositional variables are assumed to be false and are not shown

current working hypothesis. As a likely successor, the doctor decides that the patient may have a drinking problem, thus we have p_{DR} here. We leave the rest and get the state s_1.

Another likely hypothesis given s could be that we had wrong or outdated information in our history record of the patient and thus didn't see that the patient has a heart problem. In that case, we want a likely successor s_2 in which we have p_{HR}, and because of that, also p_{D}. This is shown in state s_2.

As a last set of hypotheses, we think it is conceivable, also not likely, that, the thorough examination did not notice that a tumor is present. Thus, we add p_{T} at state s_3. We could also think that the observation of the patients skin complexion might have been wrong, thus we have here $\neg p_{\mathrm{Y}}$ as a working hypothesis. This is all shown in state s_3.

From the model shown in figure 8.2 we can now determine the values for some more complicated formulas. For example, while we have $\neg p_{\mathrm{HR}}$ in s, we still have a likely successor with p_{HR}. Thus, in s we have $\mathsf{L}p_{\mathrm{HR}}$, i.e. it is likely that the patient has heart problems. We also have $\neg p_{\mathrm{HR}}$, and because $\mathscr{L}$ is reflexive also $\mathsf{L}\neg p_{\mathrm{HR}}$. This shows that $\mathsf{L}A$ and $\mathsf{L}\neg A$ can both be valid in the same state. Although p_{Y} is in every likely successor of s, it is not in every successor, thus we don't have $\mathsf{G}\, p_{\mathrm{Y}}$ in s. It's another matter for p_{HP}. This is, negated, present in all successor states of s, thus we have $\mathsf{G}\neg p_{\mathrm{HP}}$ in s.

These last two examples show that it's not the same that a formula is valid at a certain state and that is generally valid, i.e. that G of it is valid. Therefore, if, for the previous example, we want to express that it is certain that the patient does not have a tumor, we take $\mathsf{G}\, p_{\mathrm{T}}$ into our hypothesis and not simply p_{T}. While the latter allows a successor to have $\neg p_{\mathrm{T}}$ the former does not.

The example shows how a decision making process could be done using the logic of likelihood. An expert creates a model by stating a state and its hypotheses and then subsequently enlarges the model by adding all the likely and conceivable successors. This process is done until a complete model is created. Then, for each state the values for more complex formulas can be determined until an answer to the question sought is found. While this process is feasible for certain problems, it is too complicated and too time consuming to effectively use it. Furthermore, it's most of the time hard to determine what sets of hypotheses are likely and what sets are conceivable given a specific set of hypotheses. And, last but not least, the resulting decision should be valid for all models and not just for the one which is created.

Thus, to solve problems, it will be much easier to state special, non-logical axioms instead of directly working with hypotheses. For our example, this could mean that the doctor states, from his experience, that if a patient has a yellow complexion he either has a drinking problem or he has hepatitis. This could be formalized as $\mathsf{G}p_{\mathrm{Y}} \to \mathsf{G}p_{\mathrm{DR}} \vee \mathsf{G}p_{\mathrm{HP}}$. Note that we don't simply take $p_{\mathrm{Y}} \to p_{\mathrm{DR}} \vee p_{\mathrm{HP}}$, which would be much weaker.

The deductions that can be made from such a set of hypotheses and its non-logical axioms don't normally result in clear answers to a problem. Consider the earlier example and assume that it can be deduced that $\mathsf{L}\ \mathsf{G}\ p_{\mathrm{T}}$. The doctor might in that case decide it's best to operate. But what is he to do if only $\mathsf{L}^3\mathsf{G}p_{\mathrm{T}}$ can be deduced, or something even weaker. In that case, much will depend on the interpretation of the resulting formulas. This could again be formalized with extra logical axioms. For example, the doctor could include $\mathsf{LG}p_{\mathrm{T}} \rightarrow p_{\mathrm{O}}$, where p_{O} stands for an operation, to model his interpretation of the system. Such extra logical axioms are similarly problematic, though.

8.4.2 Correctness of a Protocol

The second example is much more complicated than the one of the previous section. It is the reason why Halpern and Rabin created the logic of likelihood. The main reason for the logic was to be able to verify the special protocol used in this example. This example shows that LL^- can be used to prove certain properties of a protocol used to exchange data between two persons. In this situation, credibility and threats get important, and LL^- can be used to formalize and analyze these facets as well.

We take a close look at this example after the introduction of the various calculi used to make proofs in the logic. Furthermore, we can use the LWB to show that the results for the protocol really hold. This is all shown in chapter 12.

Chapter 9

Hilbert Calculus

Semantically dealing with LL$^-$ formulas can be quite problematic, as mentioned in the previous chapter. Thus, it is often much easier to use a syntactical approach. For that reason, this chapter defines a calculus which is provably equivalent to the semantical definition of the previous chapter, but much easier to prove with.

In [22] Halpern and Rabin give a Hilbert calculus for LL$^-$. This calculus can be used to make proofs in LL$^-$ and it also shows some properties of LL$^-$ itself.

9.1 Calculus

Definition 14 ***Hilbert Calculus***

(AX1)	*instances of classical tautologies*
(AX2)	$\mathsf{G}A \to A$
(AX3)	$\mathsf{G}A \to \mathsf{GG}A$
(AX4)	$\mathsf{G}A \to \neg\mathsf{L}\neg A$
(AX5)	$A \to \mathsf{L}A$
(AX6)	$\mathsf{L}(A \vee B) \leftrightarrow (\mathsf{L}A \vee \mathsf{L}B)$
(AX7)	$\mathsf{G}(A \to B) \to (\mathsf{G}A \to \mathsf{G}B)$
(AX8)	$\mathsf{G}(A \to B) \to (\mathsf{L}A \to \mathsf{L}B)$

Inference Rules:

(R1) $\dfrac{A}{\mathsf{G}A}$ *(generalization)*

(R2) $\dfrac{A, A \to B}{B}$ *(modus ponens)*

All the axioms are axiom schemes, i.e. every formula of LL^- can be substituted for A and B. These axioms are completely the same as in [22]. Only small notational changes were made and the axioms concerning the L^* operator are left out.

A closer look at the individual axioms of the Hilbert Calculus helps to better understand the motivation behind each of them:

(AX1) This axiom makes sure that LL^- does include the classical propositional calculus. Everything that is classically provable can also be proven in LL^-.

(AX2) This axiom enforces the reflexivity of G. It can actually be deduced easily from AX4 and the contraposition of AX5. Thus, it could be left out.

(AX3) This comes from the transitivity of G. G is semantically defined over reachability, i.e. over an arbitrary number of states, thus is transitive.

(AX4) Using the previously defined abbreviations, this is actually just $\mathsf{G}A \rightarrow \mathsf{K}A$ and states that G does include K (which is based on the fact that G is defined over $\mathscr{L} \cup \mathscr{C}$ and thus includes all the relationships used for L and K).

(AX5) This expresses the reflexivity of L (it would be somewhat clearer using the abbreviation for K, then the axiom could be formulate as $\mathsf{K}A \rightarrow A$, which nicely corresponds to AX2).

(AX6) This axiom expresses that L is a modal operator. More common, but equivalent would be to use the abbreviation for K again and use the axiom $(\mathsf{K}A \wedge \mathsf{K}(A \rightarrow B)) \rightarrow \mathsf{K}B$.

(AX7) This is similar to AX6 but for the operator G.

(AX8) Finally, this axiom states that the modal operator G encompasses the modal operator L.

Furthermore, the Hilbert calculus also defines two inference rules:

(R1) This is the same generalization used in most modal logics. Similar to modal logic, this rule is not the same as an axiom $A \rightarrow \mathsf{G}A$. Such an axiom, together with (AX2) would directly lead to $A \leftrightarrow \mathsf{G}A$, which would make the operator G useless.

(R2) This is the standard inference rule used for classical propositional logic.

9.2 Properties

Before treating some properties of the Hilbert calculus, we need some more definitions.

Definition 15 ***Proofs***

We say we have a proof *of formula A, written as $\vdash_{\overline{H}} A$ if*

- *A is an instance of one of the axioms (AX1) to (AX8), or*
- *A is the conclusion of one of the inference rules and there is a proof for the premiss(es) of this rule.*

Definition 16 ***Proof Length*** $\vdash_H^n$
The length n of a proof $\vdash_H^n A$ *is inductively defined as*

- $\vdash_H^n A$ *for every* $n \geq 0$ *if* A *is an instance of one of the Axioms (AX1) to (AX8).*
- $\vdash_H^n A$ *if* A *is the conclusion of an inference rule and for each of its premisses* A_i *we have* $\vdash_H^{n_i} A_i$ *with* $n_i < n$.

The following theorem makes sure that the semantics introduced in the previous chapter are equivalent to the Hilbert calculus defined above.

Theorem 3 ***Axiom System***
The previously given Hilbert calculus is sound and complete for LL$^-$*, i.e.*

$$\models A \quad \Leftrightarrow \quad \vdash_H A$$

A proof of this theorem can be found in [22]. Because only notational changes were made to the calculus, the proof is not shown here.

Theorem 3 allows us to use this syntactical calculus instead of examining models to decide of a formula if valid or not. As we have seen in an example in the previous chapter, the treatment of models can be quite complicated and error prone. Thus, making proofs in the strictly syntactical Hilbert calculus can be much easier.

Example 5 ***Hilbert Proof***
As an example, we make in the following a small, simple proof in the Hilbert calculus. This proof shows some of the benefits and problems of Hilbert style calculi. We want to prove the formula $(\mathsf{G}A \wedge \mathsf{G}B) \leftrightarrow \mathsf{G}(A \wedge B)$ in LL$^-$. The following is a proof in the Hilbert style calculus, showing on the right side the axioms and inference rules that were used:

$\vdash_H$	$A \to (B \to (A \wedge B))$	(classical)
$\vdash_H$	$\mathsf{G}(A \to (B \to (A \wedge B)))$	(R1)
$\vdash_H$	$\mathsf{G}A \to (\mathsf{G}B \to \mathsf{G}(A \wedge B))$	(AX7 and R2)
$\vdash_H$	$(\mathsf{G}A \wedge \mathsf{G}B) \to \mathsf{G}(A \wedge B)$	(classical)
$\vdash_H$	$A \wedge B \to A$	(classical)
$\vdash_H$	$\mathsf{G}((A \wedge B) \to A)$	(R1)
$\vdash_H$	$\mathsf{G}(A \wedge B) \to \mathsf{G}A$	(AX7 and R2)
$\vdash_H$	$\mathsf{G}(A \wedge B) \to \mathsf{G}B$	(analogous)
$\vdash_H$	$\mathsf{G}(A \wedge B) \to (\mathsf{G}A \wedge \mathsf{G}B)$	(classical)

9.3 Automated Theorem Proving

There are two main problems with proofs in the Hilbert calculus. First, (AX1) states that everything which is classically provable is also provable in LL^-. Of course, we could use a classical theorem prover to decide if a given formula is classically provable. Thus, it would be quite easy for a computer to decide if the proof given above is a valid proof or not. But that is not exactly the goal of an automated theorem prover. Instead, a theorem prover should take an arbitrary formula of the logic and try to find a proof for it. But, as in the example above, the first step of the proof could be to take a classically provable formula and to go on deriving from there. Unfortunately, there are infinitely many classically provable formulas that we could use, thus if we want a decidable procedure, this solution is not feasible. Even if we are content with a semi-decidable theorem prover, such a procedure would be much too slow for practical use.

Thus, most theorem provers go a different way. Instead of starting with an axiom and then using rules to try to deduce the desired formula, they start with the formula to prove and apply the rules backwards, until an axiom is reached. This would, in a way, solve the previous problem, because we could check any formula if its provable in classical logic with an external theorem prover and if its not we'd try to apply an inference rule or axiom of the Hilbert calculus. This might work, but would be quite slow. Furthermore, there is the second problem with the Hilbert calculus. The modus ponens rule (R2) violates the so called subformula property which states that the premiss of a rule only contains subformulas of its conclusion. This makes it very hard, if not impossible, for a theorem prover to apply the rule backwards. Which formula A should be taken in the premiss of the rule. There are infinitely many formulas available, and even if we can somehow limit the number of available formulas it's presumably still too slow.

Thus, the Hilbert style calculus has several drawbacks, preventing it from being implemented on a computer in an automatic theorem prover. While the proof of theorem 2 says that there is a decision procedure—which terminates—that can be implemented on a computer, it does not use the Hilbert calculus given above. Instead it tries to create a counter model to a given formula, and if that fails, the given formula must be satisfiable, thus its negation provable. Although this solution does work, it is quite error prone for implementing. Furthermore, all the decision procedures already implemented in the Logics Workbench LWB use a syntactical approach instead of a semantical one.

Thus, we need a syntactical calculus like the Hilbert calculus given above, which does not have the same drawbacks and which can easily be implemented on a computer. This task is the contents of the next chapter.

Chapter 10

Tait Calculus

Most of the decision procedures implemented in the Logics Workbench use sequent calculi (see next chapter). But before we introduce a sequent calculus for LL^-, we start with a Tait calculus. The main reason behind this are the completeness and soundness proofs. Using intermediate calculi, instead of directly making these proofs for the calculus that will be implemented in the end, is much easier.

The Tait calculus does no more just deal with single formulas, but instead with sets of formulas. Thus, before we take a closer look at it, we need some additional notational remarks.

10.1 Notation

- We use the symbols Γ and Δ, sometimes with indices, for sets of formulas.
- Γ, Δ stands for the union of Γ and Δ and we also write Γ, A for the union of Γ and the set consisting solely of the formula A, i.e. $\Gamma, A := \Gamma \cup \{A\}$.
- We write $\bigvee \Gamma$ and $\bigwedge \Gamma$ for the disjunction, respectively the conjunction of all elements of Γ.
- We write $\neg\Gamma$ for the set consisting of all elements of Γ, with an additional leading negation.
- We write $\mathsf{F}\Gamma$ for the set consisting of all elements of Γ, with an additional leading F-operator. Similarly we write $\mathsf{G}\Gamma$, $\mathsf{K}\Gamma$, and $\mathsf{L}\Gamma$.

All formulas treated in this chapter are assumed to be in a special form, the so called negation normal form.

Definition 17 ***Negation Normal Form***
The negation normal form $\mathrm{nnf}(A)$ *of a formula A is inductively defined as:*

$$
\begin{aligned}
\mathrm{nnf}(p_i) &= p_i,\\
\mathrm{nnf}(A \wedge B) &= \mathrm{nnf}(A) \wedge \mathrm{nnf}(B),\\
\mathrm{nnf}(A \vee B) &= \mathrm{nnf}(A) \vee \mathrm{nnf}(B),
\end{aligned}
$$

$$\begin{aligned}
\text{nnf}(A \to B) &= \text{nnf}(\neg A \vee B),\\
\text{nnf}(A \leftrightarrow B) &= \text{nnf}(A \to B \wedge B \to A),\\
\text{nnf}(\mathsf{L}A) &= \mathsf{L}(\text{nnf}(A)),\\
\text{nnf}(\mathsf{K}A) &= \mathsf{K}(\text{nnf}(A)),\\
\text{nnf}(\mathsf{G}A) &= \mathsf{G}(\text{nnf}(A)),\\
\text{nnf}(\mathsf{F}A) &= \mathsf{F}(\text{nnf}(A)),\\
\text{nnf}(\neg p_i) &= \neg p_i,\\
\text{nnf}(\neg\neg A) &= \text{nnf}(A),\\
\text{nnf}(\neg A \wedge B) &= \text{nnf}(\neg A) \vee \text{nnf}(\neg B),\\
\text{nnf}(\neg A \vee B) &= \text{nnf}(\neg A) \wedge \text{nnf}(\neg B),\\
\text{nnf}(\neg \mathsf{L}A) &= \mathsf{K}(\text{nnf}(\neg A)),\\
\text{nnf}(\neg \mathsf{K}A) &= \mathsf{L}(\text{nnf}(\neg A)),\\
\text{nnf}(\neg \mathsf{G}A) &= \mathsf{F}(\text{nnf}(\neg A)),\\
\text{nnf}(\neg \mathsf{F}A) &= \mathsf{G}(\text{nnf}(\neg A)).
\end{aligned}$$

The negation normal form actually just moves all negation symbols down (if the formula is viewed as a tree) to the propositional variables. To be able to define it as easily as in the previous definition, i.e. to have negation really only in front of propositional variables, we need the dual operators $\vee$, F and K. Otherwise, nnf($\neg\mathsf{L}A$) would be defined as $\neg\mathsf{L}\neg$nnf($\neg A$). Although this is the same as Knnf($\neg A$), the latter form directly shows that negation symbols are only in front of propositional variables.

Example 6 ***Negation Normal Form***

As an example,

$$\text{nnf}(\neg\mathsf{L}(\mathsf{G}A \wedge \neg\neg\mathsf{F}\neg\mathsf{L}B)) = \mathsf{K}(\mathsf{F}\neg A \vee \mathsf{GL}B).$$

If no abbreviations for the dual operators are used, this would instead be

$$\neg\mathsf{L}\neg(\neg(\mathsf{G}A \wedge \neg\mathsf{GL}B),$$

which is not easily recognizable as being in negation normal form.

Notation $\Gamma_\star$

If Γ is a set of formulas, then $\Gamma_\star$ denotes a set of formulas with the property that

$$A \in \Gamma_\star \quad \Rightarrow \quad \star A \in \Gamma,$$

where $\star$ is one of the operator L, F, G, or K.

This notation will later be used to easily describe a special set of formulas determined by their context.

Thus, Γ_{L} is a set of formulas containing some or all of the formulas in Γ which begin with the operator L but where the leading operator is removed. Most of the time, Γ_{L} will contain all the L-formulas of Γ, but this is not required.[1]

Example 7 ***Operator Sets***
As an example, if $\Gamma = \{A, \mathsf{L}B, \mathsf{L}C, \mathsf{L}D, \mathsf{G}E, \mathsf{K}A\}$, then Γ_{L} could be, for example $\Gamma_{\mathsf{L}} = \{B, D\}$.

10.2 Calculus

Definition 18 ***Tait calculus for LL$^-$***
We define the following Tait calculus

$$\frac{}{p, \neg p, \Gamma}\ (\mathrm{id})$$

$$\frac{A, B, \Gamma}{A \vee B, \Gamma}\ (\vee) \qquad \frac{A, \Gamma \quad B, \Delta}{A \wedge B, \Gamma, \Delta}\ (\wedge)$$

$$\frac{A, \Gamma}{\mathsf{L}A, \Gamma}\ (\mathsf{L}) \qquad \frac{A, \Gamma_{\mathsf{L}}, \mathsf{F}\Gamma_{\mathsf{F}}}{\mathsf{K}A, \Gamma}\ (\mathsf{K})$$

$$\frac{A, \Gamma}{\mathsf{F}A, \Gamma}\ (\mathsf{F}) \qquad \frac{A, \mathsf{F}\Gamma_{\mathsf{F}}}{\mathsf{G}A, \Gamma}\ (\mathsf{G})$$

Later we will need an extended Tait calculus, including an additional rule, the cut rule. This rule has the following form

$$\frac{A, \Gamma \quad \neg A, \Gamma}{\Gamma}\ (\mathrm{cut})$$

Before we look at some of the properties of this calculus, we note what these rules actually mean and why they are written like that.

When we make a backward proof search with such a calculus, we do nothing else than trying to create a counter model for the formula given (cf. [29] for a close look at this situation using graph calculi). If all possible counter model constructions fail, then we know that our formula must be valid in all models, i.e. that it is provable.

We take a look at all individual rules to see how this counter model construction is done. First, we have the axiom (id). Clearly, if a state in a model is forced to make such a set of formulas true, it has to fail. It's just not possible to make p and $\neg p$ false in the same state of a model.

[1] Actually, the rules given below, if implemented on a computer, will always take all possible formulas into the set. That way, no formula that might be used later is forgotten, which could force backtracking. On the other hand, a proof is valid as well if not all formulas are taken over, if the right ones are present.

Thus, if we reach such a step, the counter model creation has failed. If all our possible ways to a counter model reach such a set of formulas (i.e. an axiom), we know that the counter model search failed and thus that the formula must be provable.

The ($\vee$)-rule tries to make $A \vee B$ false, thus must try to make both formula false. Thus, in the Tait calculus the comma between formulas can be interpreted as '$\vee$'.

The ($\wedge$)-rule, on the other hand, wants to make $A \wedge B$ false, and that can be done in either of two ways, either by making A false or by making B false. These two possibilities are expressed with the two premisses of the rule. We have to try both possible paths to be sure that no counter model can be created. Thus, only if both of the branches end in axioms, we know the formula is provable.

These first three rules can directly been taken over from a Tait calculus for classical logic, and actually make sure that the calculus will include the whole classical propositional logic. As we will see later, the axiom given in the calculus could also have been given using full formulas instead of only propositional variables. Using the definition above makes the following proofs a little bit simpler, thus we stick to that for the time being.

The remaining four rules are the more interesting ones, i.e. the ones that define the modal operators. As mentioned before, all the rules assume that the formulas are in negation normal form. Because the negation normal form uses four modal operators, we also need special rules for all of these.[2]

The (L)-rule simply expresses that the L-operator is reflexive. Thinking of counter model creation, we see that if we want to make the formula $\mathsf{L}A$ false, we need to make A false in the current state. This those not make sure that $\mathsf{L}A$ is false. But if A is true in the current state, we surely have $\mathsf{L}A$. Additionally, we need to make sure, that A is also false in all states $\mathscr{L}$-reachable from the current one. This is done in the (G)- and (K)-rules (see below).

The rule for F is dual to the rule for L. Limited to the current state, L and F are, for counter model creation, the same, as both are reflexive. The difference between these two operators will come into play if we examine the formulas in states reachable from the current one. While it is enough to treat $\mathscr{L}$-reachable states for L, we need to treat the $\mathscr{C}$-reachable (or, including $\mathscr{L}$, the reachable) states for F.

A bit more complicated are the rules for K and G. As mentioned above, they not only have to treat their respective operator but also the cases not yet treated for L and F. We first take a look at the rule for G, because is a little bit simpler. If we want to create a counter model for $\mathsf{G}A$ its easiest to create a $\mathscr{C}$-successor to the current state in which A is made false. We could do that with a rule like

$$\frac{A, \Gamma}{\mathsf{G}A, \Gamma}\ (\mathsf{G})\ ,$$

[2] But because of the negation normal form we don't need to treat negations, except in axioms.

because we also want Γ to be made false. This way we end up with an incorrect rule, i.e. if the premiss of the rule is valid the conclusion does not need to be as well.[3] Furthermore, we want to make Γ to be false in the *current* state, not in a successor. Thus we could take the rule

$$\frac{A}{\mathsf{G}A, \Gamma}\ (\mathsf{G})\ ,$$

which would be correct. But this rule leaves out the conditions mentioned above for (F). We remarked that, when making a formula F false, we need to make it false in the current state but also in all successor state. That holds for the state created with this rule as well. Thus, we have to take over all the F-formulas into the new state as well, to try to make them false. Because the F operator is transitive, we have to take over the whole F formula and not just the formula without F. Thus we need not only be able to make the formula false in the successor but also in its successor, and so on. Thus, we end up with the rule

$$\frac{A, \mathsf{F}\Gamma_{\mathsf{F}}}{\mathsf{G}A, \Gamma}\ (\mathsf{G})\ ,$$

i.e. we take over all (or at least the necessary ones) F formulas into the new state.

Even more complicated is the last rule, the rule for the K operator. A first idea might be to use the same arguments as for G and use the rule

$$\frac{A, \mathsf{F}\Gamma_{\mathsf{F}}}{\mathsf{K}A, \Gamma}\ (\mathsf{K})\ .$$

But to make the formula KA false in a state, we need to create an $\mathscr{L}$-successor where A is false, not a $\mathscr{C}$successor as for G. For an $\mathscr{L}$-successor we have to take care of the condition expressed with the (L)-rule. We noted that to make LA false in a state, we also have to make A false in all $\mathscr{L}$-successor states. Therefore, we have to take over the L-formulas as well. This time though, because the L operator is not transitive, we only have to take over B if LB was part of Γ. Thus, we end up with the rule

$$\frac{A, \Gamma_{\mathsf{L}}, \mathsf{F}\Gamma_{\mathsf{F}}}{\mathsf{K}A, \Gamma}\ (\mathsf{K})\ ,$$

as used in the definition above.

If we compare the rules of this calculus with existing modal logics, like KT or S_4 (as for example in [29]), we note some similarities. In KT or S_4, which are also reflexive, the rule for the $\Diamond$-operator is

[3] for example, if $A \vee B$ is valid for a model, G$A \vee B$ does not have to be as well.a

$$\frac{A, \Gamma}{\Diamond A, \Gamma}\ (\Diamond)\ .$$

This corresponds nicely with the rules we used for L and F above.

The rule for $\Box$ in KT is, for example

$$\frac{A, \Gamma_{\Diamond}}{\Box A, \Gamma}\ (\Box_{\mathsf{KT}})\ .$$

where as in S_4 the rule is

$$\frac{A, \Diamond\Gamma_{\Diamond}}{\Box A, \Gamma}\ (\Box_{\mathsf{S}_4})\ .$$

The latter rule is similar to the rule we used above for G, which is quite clear, because only S_4 is transitive while KT is not. The rule for K, on the other hand, is some sort of mixture of the $\Box$-rules of KT and S_4, clearly because the L-operator is reflexive but not transitive, as the $\Box$-operator in KT, whereas the G-operator is reflexive and transitive, like the $\Box$-operator in S_4.

10.3 Correctness

A first step to show that the newly introduced Tait calculus is equivalent to the previously defined Hilbert calculus, is to show that the Tait calculus is correct, i.e. that if we have a proof of a formula in the Tait calculus, we can find a prove in the Hilbert calculus as well. As mentioned, a proof of a set of formulas is interpreted as having a proof of the disjunction of all formulas of the set.

Definition 19 ***Tait Proof***
We say we have a proof *for a formula A in the Tait calculus, written as* $\vdash_{T} A$ *if*

- *A is an instance of the axiom (id), or*
- *A is the conclusion of one of the rules ($\vee$), ($\wedge$), (L), (F), (K), or (G) and there are proofs for all premisses of this rule.*

We say we have a proof *for a formula A in the Tait calculus with cut, written as* $\vdash_{T+C} A$ *if A is either one of the above or additionally the conclusion of a (cut)-Rule, where proofs exists for both premisses.*

Definition 20 ***Proof Length*** $\vdash^{n}_{T}$
The length *n of a* proof $\vdash^{n}_{T} A$ *is inductively defined as*

- $\vdash^{n}_{T} A$ *for every $n \geq 0$ if A is an instance of the axiom (id).*

- *$\vdash^{n}_{T} A$ if A is the conclusion of an inference rule and for each of its premisses A_i we have $\vdash^{n_i}_{T} A_i$ with $n_i < n$.*

Similarly we define $\vdash^{n}_{T+C} A$.

This definition is the same as the definition for the proof length of proofs in the Hilbert style calculus. It just says that if a formula is provable with a length of n, then the longest branch in the proof tree will have at most length n, but may have less.

Definition 21 ***Main and Side Formulas***
The main formula *of a rule is the formula that appears distinguished in the conclusions of the rules in the calculus shown above.*

All other formulas of the conclusion are called side formulas.

Example 8 ***Main Formula***
As an example, $A \vee B$ is the main formula in the ($\vee$)-rule and $\mathsf{G}A$ is the main formula of the (G)-rule.

Theorem 4 ***Correctness***

$$\vdash_{T} \Gamma \quad \Rightarrow \quad \vdash_{H} \bigvee \Gamma$$

Proof
We transform every proof in the Tait calculus into a proof in the Hilbert calculus. Because the Hilbert calculus is correct, the Tait calculus will thus be as well.

The proof is the done by induction over the length of the proof in the Tait calculus. Thus, if $\vdash^{n}_{T} \Gamma$ then we make an induction on n to show that $\vdash_{H} \bigvee \Gamma$, by transforming the proof as follows:

$n = 0$ Then Γ is an axiom, i.e. a conclusion of the (id)-rule.

In that case, $\bigvee \Gamma$ is a classical tautology and because of (AX1) provable in the Hilbert calculus.

$n > 0$ We distinguish different cases, depending on the last rule used in the proof of Γ. If Γ is an axiom, then the assumption follows as in the case of $n = 0$.

Thus let $\Gamma = A, \Gamma'$, where A is the main formula of the last rule used in the proof.

($\vee$) $A \equiv B \vee C$.
Applying the ($\vee$)-rule backwards gives us $\vdash^{<n}_{T} B, C, \Gamma'$. From the induction hypothesis we know that $\vdash_{H} \bigvee B, C, \Gamma'$. But then, by a simple classical transformation also $\vdash_{H} \bigvee B \vee C, \Gamma',$.

($\wedge$) $A \equiv B \wedge C$.

With the ($\wedge$)-rule backwards we get $\vdash^{\leq n}_{T} B, \Gamma'$ and $\vdash^{\leq n}_{T} C, \Gamma'$. With the induction hypothesis this gives us $\vdash_{H} \bigvee B, \Gamma'_1$ and $\vdash_{H} \bigvee C, \Gamma'_2$. From the classical tautology

$$(B \vee \bigvee \Gamma'_1) \to ((C \vee \bigvee \Gamma'_2) \to ((B \wedge C) \vee (\bigvee \Gamma'_1, \Gamma'_2)))$$

and with modus ponens (R2) follows $\vdash_{H} \bigvee B \wedge C, \Gamma'$.

(L) $A \equiv \mathsf{L}B$.

We know from the induction hypothesis and the premiss of the (L)-rule that $\vdash_{H} \bigvee A, \Gamma'$. With AX5, the classical tautology

$$(A \vee \bigvee \Gamma') \wedge (A \to \mathsf{L}A) \to (\mathsf{L}A \vee \bigvee \Gamma')$$

and modus ponens follows $\vdash_{H} \mathsf{L}A, \Gamma'$.

(K) $A \equiv \mathsf{K}B$.

Using the (K)-rule backwards and the induction hypothesis follows $\vdash_{H} \bigvee A, \Gamma_{\mathsf{L}}, \mathsf{F}\Gamma_{\mathsf{F}}$. With generalization (R1) follows $\vdash_{H} \mathsf{G}(A \vee \bigvee \Gamma_{\mathsf{L}}, \mathsf{F}\Gamma_{\mathsf{F}})$. This is, with some rewriting $\vdash_{H} \mathsf{G}(\neg A \to (\bigvee \Gamma_{\mathsf{L}}, \mathsf{F}\Gamma_{\mathsf{F}}))$. With (AX8) and (R2) follows $\vdash_{H} \mathsf{L}\neg A \to \mathsf{L}(\bigvee \Gamma_{\mathsf{L}}, \mathsf{F}\Gamma_{\mathsf{F}})$. With (AX6) and some rewriting we get $\vdash_{H} \mathsf{K}A \vee \bigvee \mathsf{L}\Gamma_{\mathsf{L}}, \mathsf{L}\mathsf{F}\Gamma_{\mathsf{F}}$. From $\mathsf{L}C \to \mathsf{F}C$ and $\mathsf{F}\mathsf{F}C \to \mathsf{F}C$ follows $\vdash_{H} \bigvee \mathsf{K}A, \mathsf{L}\Gamma_{\mathsf{L}}, \mathsf{F}\Gamma_{\mathsf{F}}$ and thus surely $\vdash_{H} \bigvee \mathsf{K}A, \mathsf{L}\Gamma_{\mathsf{L}}, \mathsf{F}\Gamma_{\mathsf{F}}, \Gamma'$. [4]

(F) $A \equiv \mathsf{F}B$.

With the (F)-rule backwards and the induction hypothesis follows $\vdash_{H} A \vee \bigvee \Gamma'$. Because of $\vdash_{H} A \to \mathsf{F}A$ follows $\vdash_{H} \mathsf{F}A, \Gamma'$.

(G) $A \equiv \mathsf{G}B$.

With the (G)-rule backwards and the induction hypothesis follows $\vdash_{H} A \vee \bigvee \mathsf{F}\Gamma_{\mathsf{F}}$. This is the same as $\vdash_{H} \neg \bigvee \mathsf{F}\Gamma_{\mathsf{F}} \to A$. With generalization (R1) follows $\vdash_{H} \mathsf{G}(\neg \bigvee \mathsf{F}\Gamma_{\mathsf{F}} \to A)$. Using (AX7) this is $\mathsf{G}\neg \bigvee \mathsf{F}\Gamma_{\mathsf{F}} \to \mathsf{G}A$. Translated back this is $\mathsf{F} \bigvee \mathsf{F}\Gamma_{\mathsf{F}} \vee \mathsf{G}A$. From $\mathsf{F}\mathsf{F}C \to \mathsf{F}C$ and $\mathsf{F}(A \vee B) \to (\mathsf{F}A \vee \mathsf{F}B)$ follows $\vdash_{H} \bigvee \mathsf{F}\Gamma_{\mathsf{F}} \vee \mathsf{G}A$. This surely leads to $\vdash_{H} \bigvee \mathsf{G}A, \Gamma'$.

10.4 Completeness

Completeness of the Tait calculus cannot be proven as easily as correctness. Before we can start to prove the completeness theorem, we need some auxiliary lemmas.

[4] This shows that the (K)-rule cannot be written similar for L and F. While we have $\mathsf{F}\mathsf{F}A \to \mathsf{F}A$, i.e. F is transitive, we do not have the same for L, which is not transitive. If we put L in the premiss for the L-formulas, the rule would not be correct.

Lemma 1 ***Weakening***

$$\vdash^{n}_{T} \Gamma \quad \Rightarrow \quad \vdash^{n}_{T} \Gamma, \Delta$$

Proof
We make the proof by induction on the length n of the proof:

$n = 0$ then Γ is an axiom.
In that case, Γ, Δ is an axiom as well, thus $\vdash^{0}_{T} \Gamma, \Delta$

$n > 0$ If Γ is an axiom, then Γ, Δ as well.
If Γ is not an axiom, then Γ was deduced with one of the rules of the Tait calculus. We assume that $\Gamma = A, \Gamma'$, where A is the main formula of the last step of the proof.
We now distinguish different cases, depending on the last rule used in the proof:

- ($\vee$) By applying the induction hypothesis to the premiss and then carrying through the ($\vee$)-step we get the assumption.
- ($\wedge$) By applying the induction hypothesis on one of the premisses and then carrying through the ($\wedge$)-step we get the assumption.
- (L) This too easily follows from the induction hypothesis and the premiss.
- (K) The (K)-rule has built-in weakening. Thus, from the premiss of the last step we directly get the assumption by expanding Γ' as necessary.
- (F) Again, this easily follows form the induction hypothesis and the premiss.
- (G) As the (K)-rule, the (G)-rule does have built in weakening as well. Thus, the assumption directly follows from the premiss.

Lemma 2 ***Cut***
From $\vdash_{T} \Gamma, A$ *and* $\vdash_{T} \Delta, \neg A$ *follows* $\vdash_{T} \Gamma, \Delta$.

Proof
We make the proof first by an induction on the construction of A, i.e. the length n of A.

$n = 0$ $A \equiv p_i$
We now make a subinduction on the length m of the proof of Γ, A:

$m = 0$ Γ, A is an axiom
We can distinguish two cases. If $\neg A \in \Gamma$, then we have $\Delta, \neg A \subset \Gamma, \Delta$. Thus from $\vdash_{T} \Delta, \neg A$ follows with lemma 1 (weakening) that $\vdash_{T} \Gamma, \Delta$.
Otherwise, $\neg A \notin \Gamma$. In that case, Γ is an axiom by itself. But then, Γ, Δ is an axiom as well.

$m > 0$ Γ, A needs not to be an axiom

If Γ, A is an axiom, we have the same case as for $m = 0$.

If Γ, A is not an axiom and because A is just a propositional variable, A cannot be the main formula of the last step of the proof. Nevertheless, we make a distinction between the last rule used in the proof of Γ, A.

In the case of the rules ($\vee$), ($\wedge$), (L) and (F) we can use the induction hypothesis on the premiss together with $\models_T \Delta, \neg A$ to get a new premiss without any A. We then carry through the same deduction step to get the desired result.

If the last rule was either a (K) or (G)-rule, then A cannot appear in the premiss, because it does not start with an L or F operator. Thus, from the same premiss, it is also possible to deduce Γ, Δ, using the built-in weakening of the rule and leaving out A.

$n > 0$ We make an induction on the sum of the lengths m_1 of the proofs of Γ, A and of m_2 of $\Delta, \neg A$.

$m_1 = 0$ Γ, A is an axiom

This is equal to the case that A is a propositional variable.

$m_2 = 0$ $\Delta, \neg A$ is an axiom

Again, this is equal to the case that A is propositional variable.

$m_1 > 0$ If either Γ, A or $\Delta, \neg A$ are axioms, then we argue as for $m_1 = 0$ or $m_2 = 0$
$m_2 > 0$ respectively. Thus we assume that Γ, A and $\Delta, \neg A$ are both not axioms

We now have to distinguish different cases, depending on the last rules of the proofs and their main formulas. Because—in this special case—it is quite easy to miss one of the possible cases, we list all 16 possible combinations below, even though several of them are quite similar.

1) A was not the main formula, $\neg A$ was not the main formula and both proofs did not end with a (K)- or (G)-rule:
In this case we use the hypothesis of the subinduction on the premiss of the last step of $\models_T \Gamma, A$ together with $\models_T \Delta, \neg A$ to get the desired result.

2) A was not the main formula, $\neg A$ was not the main formula and both proofs ended with a (K)- or (G)-rule:
If A does not start with either L or F, then it cannot be present in the premiss of the rule. Thus, by using the premiss of the proof of Γ, A we can also directly proof Γ, Δ by using the built-in weakening of the rule and leaving out A.
If A does start with either L or F, then $\neg A$ on the other hand does not. Thus the assumption directly follows from the premiss of the last step of $\models_T \Delta, \neg A$.

3) A was not the main formula, $\neg A$ was not the main formula, $\models_T \Gamma, A$ ended with a (K)- or (G)-rule but $\models_T \Delta, \neg A$ did not:
This time we use the hypothesis of the subinduction on the premiss of the last step of $\models_T \Delta, \neg A$ together with $\models_T \Gamma, A$ to get the desired result.

4) A was not the main formula, $\neg A$ was not the main formula, $\models_T \Gamma, A$ ended not with a (K)- or (G)-rule but $\models_T \Delta, \neg A$ did:
This is the same as case 1.

5) A was the main formula, $\neg A$ was not the main formula and both proofs did not end with a (K)- or (G)-rule:
This can be done as case 3.

6) A was the main formula, $\neg A$ was not the main formula and both proofs ended with a (K)- or (G)-rule:
This case is a bit more complicated. We have to distinguish several subcases:
   - $A \equiv \mathsf{K}B$ and the main formula of $\models_T \Delta, \mathsf{L}\neg B$ is $\mathsf{K}C$
     Then the premisses of the last steps give us $\models_T B, \Gamma_\mathsf{L}, \mathsf{F}\Gamma_\mathsf{F}$ respectively $\models_T \neg B, \Delta'_\mathsf{L}, \mathsf{F}\Delta'_\mathsf{F}, C$. We can now use the hypothesis of the main induction on B to get $\models_T \Gamma_\mathsf{L}, \mathsf{F}\Gamma_\mathsf{F}, \Delta'_\mathsf{L}, \mathsf{F}\Delta'_\mathsf{F}, C$. Now, we carry through the (K)-rule on C with the appropriate weakenings to get the desired proof.
   - $A \equiv \mathsf{K}B$ and the main formula of $\models_T \Delta, \mathsf{L}\neg B$ is $\mathsf{G}C$
     In this case, $\neg B$ does not appear in the premiss of the last step of $\models_T \Delta, \mathsf{L}\neg B$. Thus with the right weakenings we can directly obtain the desired proof.
   - $A \equiv \mathsf{G}B$ and the main formula of $\models_T \Delta, \mathsf{F}\neg B$ is $\mathsf{K}C$
     In that case, the premiss of the proof of $\models_T \Delta, \mathsf{F}\neg B$ is $\mathsf{F}\neg B, \Delta'_\mathsf{L}, \mathsf{F}\Delta'_\mathsf{F}, C$. Furthermore, we can change the last step of the proof of $\models_T \mathsf{G}B, \Gamma$ to get from the premiss $\models_T B, \mathsf{F}\Gamma_\mathsf{F}$ just to $\models_T \mathsf{G}B, \mathsf{F}\Gamma_\mathsf{F}$ by not using the built-in weakening. This does not affect proof length and thus we can now use the hypothesis of the subinduction together with $\models_T \mathsf{F}\neg B, \Delta'_\mathsf{L}, \mathsf{F}\Delta'_\mathsf{F}, C$ to obtain $\mathsf{F}\Gamma_\mathsf{F}, \Delta'_\mathsf{L}, \mathsf{F}\Delta'_\mathsf{F}, C$. Now, we just carry through a (K)-step on C again to get the proof we are looking for.
   - $A \equiv \mathsf{G}B$ and the main formula of $\models_T \Delta, \mathsf{F}\neg B$ is $\mathsf{G}C$
     This case can be handled similar to the one before.

7) A was the main formula, $\neg A$ was not the main formula, $\models_T \Gamma, A$ ended with a (K)- or (G)-rule but $\models_T \Delta, \neg A$ did not:
This is the same as case 3.

8) A was the main formula, $\neg A$ was not the main formula, $\models_T \Gamma, A$ ended not with a (K)- or (G)-rule but $\models_T \Delta, \neg A$ did:
Because A is the main formula of the last step, which is not a (K)- or (G)-rule, A cannot begin with either G or K. Thus $\neg A$ cannot begin with L or F. Because $\neg A$ was not the main formula of the last step of its proof, which was a (K)- or (G)-rule, $\neg A$ cannot appear in its premiss. Thus, from the same premiss we can directly obtain our desired proof by using the built-in weakening.

9) A was not the main formula, $\neg A$ was the main formula and both proofs did not end with a (K)- or (G)-rule:
This is the same as case 1.

10) A was not the main formula, $\neg A$ was the main formula and both proofs ended with a (K)- or (G)-rule:
By switching the roles of A and $\neg A$ this case is similar to case 6.

11) A was not the main formula, $\neg A$ was the main formula, $\models_T \Gamma, A$ ended with a (K)- or (G)-rule but $\models_T \Delta, \neg A$ did not:
By switching the roles of A and $\neg A$ this case is similar to case 8.

12) A was not the main formula, $\neg A$ was the main formula, $\vdash_{T} \Gamma, A$ ended not with a (K)- or (G)-rule but $\vdash_{T} \Delta, \neg A$ did:
This is the same as case 1.

13) A was the main formula, $\neg A$ was the main formula and both proofs did not end with a (K)- or (G)-rule:
Depending on the structure of A we have to distinguish two cases:

   $\wedge$ $A \equiv B \wedge C$, and $\neg A \equiv \neg B \vee \neg C$
   From the premisses of the last steps we have $\vdash_{T}^{<m_1} \Gamma_1, B \wedge C, B$, and $\vdash_{T}^{<m_1} \Gamma_2, B \wedge C, C$ and $\vdash_{T}^{<m_2} \Delta, \neg B \vee \neg C, \neg B, \neg C$. We can use the hypothesis of the subinduction to get $\vdash_{T} \Gamma_1, B, \Delta$, $\vdash_{T} \Gamma_2, C, \Delta$, and $\vdash_{T} \Gamma, \Delta, \neg B, \neg C$. We can now use the induction hypothesis of the main induction with Γ_1, Δ, B and $\Delta, \neg B, \neg C$ to get $\vdash_{T} \Gamma_1, \Delta, \neg C$. This, together with Γ_2, Δ, C and the main induction hypothesis yields $\vdash_{T} \Gamma, \Delta$.

   $\vee$ $A \equiv B \vee C$, and $\neg A \equiv \neg B \wedge \neg C$
   with a swap of A and $\neg A$ this is equivalent to the previous case

14) A was the main formula, $\neg A$ was the main formula and both proofs ended with a (K)- or (G)-rule:
This case can actually never occur. If A is the main formula of a (K)- or (G)-rule it has to start with G or K. But then $\neg A$ has to start either with F or L and thus cannot be the main formula of a (K)- or (G)-rule.

15) A was the main formula, $\neg A$ was the main formula, $\vdash_{T} \Gamma, A$ ended with a (K)- or (G)-rule but $\vdash_{T} \Delta, \neg A$ did not:
Again we distinguish several different cases, depending on the structure of A:

   L $A \equiv \mathsf{L}B$, and $\neg A \equiv \mathsf{K}\neg B$
   From the premisses of the last steps of the proofs we get $\vdash_{T}^{<m_1} B, \mathsf{L}B, \Gamma$ and $\vdash_{T}^{<m_2} \neg B, \Delta_{\mathsf{L}}, \mathsf{F}\Delta_{\mathsf{F}}$. Using the subinduction with the sets $B, \mathsf{L}B, \Gamma$ and $\mathsf{K}\neg B, \Delta$ yields $\vdash_{T} B, \Gamma, \Delta$.
   With the main induction hypothesis we get $\vdash_{T} \Gamma, \Delta, \Delta_{\mathsf{L}}, \mathsf{F}\Delta_{\mathsf{F}}$. Now, we use one (L)-rule for every element in Δ_{L} and get $\vdash_{T} \Gamma, \Delta, \mathsf{L}\Delta_{\mathsf{L}}, \mathsf{F}\Delta_{\mathsf{F}}$. This sequent is actually the same as Γ, Δ.

   K $A \equiv \mathsf{K}B$, and $\neg A \equiv \mathsf{L}\neg B$
   With a swap of A and $\neg A$ this is equivalent to the previous case.

   F $A \equiv \mathsf{F}B$, and $\neg A \equiv \mathsf{G}\neg B$
   From the premisses of the last steps of the proof we get $\vdash_{T}^{<m_1} B, \mathsf{F}B, \Gamma$ and $\vdash_{T}^{<m_2} \neg B, \mathsf{F}\Delta_{\mathsf{F}}$. With the subinduction hypothesis and the assumption $\mathsf{G}\neg B, \Delta$, we get $\vdash_{T} B, \Gamma, \Delta$.
   Using the main induction on B, Γ, Δ and $\neg B, \mathsf{F}\Delta_{\mathsf{F}}$, yields $\vdash_{T} \Gamma, \Delta, \mathsf{F}\Delta_{\mathsf{F}}$. This sequent is actually the same as Γ, Δ, thus we have $\vdash_{T} \Gamma, \Delta$.

   G $A \equiv \mathsf{G}B$, and $\neg A \equiv \mathsf{F}\neg B$
   With a swap of A and $\neg A$ this is equivalent to the previous case.

16) A was the main formula, $\neg A$ was the main formula, $\vdash_{T} \Gamma, A$ ended not with a (K)- or (G)-rule but $\vdash_{T} \Delta, \neg A$ did:
With inverse roles for A and $\neg A$ this is the same as case 15.

Lemma 3 ***Extended Axioms***
For all formulas A and alls sets of formulas Γ we have

$$\vdash_{T} \Gamma, A, \neg A$$

Proof
We make an induction on the length n of A:

$n = 0$: then $A \equiv p_i$, i.e. A is a propositional variable.
In that case, $\Gamma, A, \neg A$ is an axiom of the Tait calculus and thus provable.

$n > 0$: we distinguish several cases depending on the structure of A.

- $A \equiv B \vee C$ and $\neg A \equiv \neg B \wedge \neg C$.
  From the induction hypothesis we get proofs for $\Gamma, B, C, \neg B$ and $\Gamma, B, C, \neg C$. We use these proofs to get the following proof:

$$\dfrac{\dfrac{\dfrac{}{\Gamma,B,C,\neg B}\text{(induction)} \quad \dfrac{}{\Gamma,B,C,\neg C}\text{(induction)}}{\Gamma,B,C,\neg B\wedge\neg C}(\wedge)}{\Gamma,B\vee C,\neg B\wedge\neg C}(\vee)$$

- $A \equiv B \wedge C$ and $\neg A \equiv \neg B \vee \neg C$.
  From the induction hypothesis we get proofs for $\Gamma, B, \neg B, \neg C$ and $\Gamma, C, \neg B, \neg C$. We use these proofs to get the following proof:

$$\dfrac{\dfrac{\dfrac{}{\Gamma,B,\neg B,\neg C}\text{(induction)} \quad \dfrac{}{\Gamma,C,\neg B,\neg C}\text{(induction)}}{\Gamma,B\wedge C,\neg B,\neg C}(\wedge)}{\Gamma,B\wedge C,\neg B\vee\neg C}(\vee)$$

Theorem 5 ***Completeness***
$\vdash_{H} A \quad \Rightarrow \quad \vdash_{T} A$

Proof
Let $\vdash^{n}_{H} A$. We now prove—with induction on the length n of the proof—that $\vdash_{T} A$.

$n = 0$ For each axiom of the Hilbert calculus we have to find a proof in the Tait calculus.

AX1 Because the Tait calculus contains the derivation rules of classical propositional logic, A will be provable.

AX2 $A \equiv \mathsf{G}B \to B$

$$\dfrac{\dfrac{\dfrac{}{\neg B,B}\,(\text{lemma 3})}{\mathsf{F}\neg B,B}\,(\mathsf{F})}{\mathsf{F}\neg B\vee B}\,(\vee)$$

AX3 $A \equiv \mathsf{G}B \to \mathsf{GG}B$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg B,B}\,(\text{lemma 3})}{\mathsf{F}\neg B,B}\,(\mathsf{F})}{\mathsf{F}\neg B,\mathsf{G}B}\,(\mathsf{G})}{\mathsf{F}\neg B,\mathsf{GG}B}\,(\mathsf{G})}{\mathsf{F}\neg B\vee\mathsf{GG}B}\,(\vee)$$

AX4 $A \equiv \mathsf{G}B \to \mathsf{K}B$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg B,B}\,(\text{lemma 3})}{\mathsf{F}\neg B,B}\,(\mathsf{F})}{\mathsf{F}\neg B,\mathsf{K}B}\,(\mathsf{K})}{\mathsf{F}\neg B\vee\mathsf{K}B}\,(\vee)$$

AX5 $A \equiv B \to \mathsf{L}B$

$$\dfrac{\dfrac{\dfrac{}{\neg B,B}\,(\text{lemma 3})}{\neg B,\mathsf{L}B}\,(\mathsf{L})}{\neg B\vee\mathsf{L}B}\,(\vee)$$

AX6 $A \equiv \mathsf{L}(B \vee C) \leftrightarrow (\mathsf{L}B \vee \mathsf{L}C)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg B,B,C}\,(\text{lemma 3}) \quad \dfrac{}{\neg C,B,C}\,(\text{lemma 3})}{\neg B\wedge\neg C,B,C}\,(\wedge)}{\mathsf{K}(\neg B\wedge\neg C),\mathsf{L}B,\mathsf{L}C}\,(\mathsf{K})}{\mathsf{K}(\neg B\wedge\neg C),\mathsf{L}B\vee\mathsf{L}C}\,(\vee)}{\mathsf{K}(\neg B\wedge\neg C)\vee(\mathsf{L}B\vee\mathsf{L}C)}\,(\vee) \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg B,B,C}\,(\text{lemma 3})}{\neg B,B\vee C}\,(\vee)}{\mathsf{K}\neg B,\mathsf{L}(B\vee C)}\,(\mathsf{K}) \quad \dfrac{\dfrac{\dfrac{}{\neg C,B,C}\,(\text{lemma 3})}{\neg C,B\vee C}\,(\vee)}{\mathsf{K}\neg C,\mathsf{L}(B\vee C)}\,(\mathsf{K})}{\mathsf{K}\neg B\wedge\mathsf{K}\neg C,\mathsf{L}(B\vee C)}\,(\wedge)}{(\mathsf{K}\neg B\wedge\mathsf{K}\neg C)\vee\mathsf{L}(B\vee C)}\,(\vee)}{(\mathsf{K}(\neg B\wedge\neg C)\vee(\mathsf{L}B\vee\mathsf{L}C))\wedge((\mathsf{K}\neg B\wedge\mathsf{K}\neg C)\vee\mathsf{L}(B\vee C))}\,(\wedge)$$

AX7 $A \equiv \mathsf{G}(B \to C) \to (\mathsf{G}B \to \mathsf{G}C)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B,\neg B,C}\,(\text{lemma 3}) \quad \dfrac{}{\neg C,\neg B,C}\,(\text{lemma 3})}{B\wedge\neg C,\neg B,C}\,(\wedge)}{\mathsf{F}B\wedge\neg C,\neg B,C}\,(\mathsf{F})}{\mathsf{F}B\wedge\neg C,\mathsf{F}\neg B,C}\,(\mathsf{F})}{\mathsf{F}(B\wedge\neg C),\mathsf{F}\neg B,\mathsf{G}C}\,(\mathsf{G})}{\mathsf{F}(B\wedge\neg C),\mathsf{F}\neg B\vee\mathsf{G}C}\,(\vee)}{\mathsf{F}(B\wedge\neg C)\vee(\mathsf{F}\neg B\vee\mathsf{G}C)}\,(\vee)$$

AX8 $A \equiv \mathsf{G}(B \to C) \to (\mathsf{L}B \to \mathsf{L}C)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B,\neg B,C}\text{(lemma 3)}\qquad\dfrac{}{\neg C,\neg B,C}\text{(lemma 3)}}{B\wedge\neg C,\neg B,C}(\wedge)}{\mathsf{F}(B\wedge\neg C),\neg B,C}(\mathsf{F})}{\mathsf{F}(B\wedge\neg C),\mathsf{K}\neg B,\mathsf{L}C}(\mathsf{K})}{\mathsf{F}(B\wedge\neg C),(\mathsf{K}\neg B\vee\mathsf{L}C)}(\vee)}{\mathsf{F}(B\wedge\neg C)\vee(\mathsf{K}\neg B\vee\mathsf{L}C)}(\vee)$$

$n > 0$ R1 $A \equiv \mathsf{G}B$ and $\mathsf{G}B$ has been deduced in the last step by (R1)

With the induction hypothesis we have $\vdash_T^{<n} B$. With the (G)-rule immediately follows $\vdash_T \mathsf{G}B$, i.e. $\vdash_T A$.

R2 $A \equiv C$ and C has been deduced in the last step by (R2) from $B \to C$ and B.

With the induction hypothesis we get $\vdash_T^{<n} \neg B \vee C$ and $\vdash_T^{<n} B$. The last step in the proof of $\neg B \vee C$ must have been a ($\vee$)-rule, thus we can also get $\vdash_T \neg B, C$.

Now we can use lemma 2 (cut) to get $\vdash_T C$, i.e. $\vdash_T A$.

Example 9 ***Tait Proof***

We now take a look at an example of a proof in the Tait calculus. We take the same formula used for the example in the Hilbert calculus (cf. 5).

The proof of the formula $(\mathsf{G}A \wedge \mathsf{G}B) \leftrightarrow \mathsf{G}(A \wedge B)$ in the Tait calculus looks as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\boldsymbol{\neg A},\neg B,\boldsymbol{A}}\text{(id)}\quad\dfrac{}{\neg A,\boldsymbol{\neg B},\boldsymbol{B}}\text{(id)}}{\neg A,\neg B,\boldsymbol{A\wedge B}}(\wedge)}{\neg A,\boldsymbol{\mathsf{F}\neg B},A\wedge B}(\mathsf{F})}{\boldsymbol{\mathsf{F}\neg A},\mathsf{F}\neg B,A\wedge B}(\mathsf{F})}{\mathsf{F}\neg A,\mathsf{F}\neg B,\boldsymbol{\mathsf{G}(A\wedge B)}}(\mathsf{G})}{\boldsymbol{\mathsf{F}\neg A\vee\mathsf{F}\neg B},\mathsf{G}(A\wedge B)}(\vee)}{\boldsymbol{(\mathsf{F}\neg A\vee\mathsf{F}\neg B)\vee\mathsf{G}(A\wedge B)}}(\vee)\quad\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\boldsymbol{\neg A},\neg B,\boldsymbol{A}}\text{(id)}}{\boldsymbol{\neg A\vee\neg B},A}(\vee)}{\boldsymbol{\mathsf{F}(\neg A\vee\neg B)},A}(\mathsf{F})}{\mathsf{F}(\neg A\vee\neg B),\boldsymbol{\mathsf{G}A}}(\mathsf{G})\quad\dfrac{\dfrac{\dfrac{\dfrac{}{\neg A,\boldsymbol{\neg B},\boldsymbol{B}}\text{(id)}}{\boldsymbol{\neg A\vee\neg B},B}(\vee)}{\boldsymbol{\mathsf{F}(\neg A\vee\neg B)},B}(\mathsf{F})}{\mathsf{F}(\neg A\vee\neg B),\boldsymbol{\mathsf{G}B}}(\mathsf{G})}{\mathsf{F}(\neg A\vee\neg B),\boldsymbol{\mathsf{G}A\wedge\mathsf{G}B}}(\wedge)}{\boldsymbol{\mathsf{F}(\neg A\vee\neg B)\vee(\mathsf{G}A\wedge\mathsf{G}B)}}(\vee)}{\boldsymbol{((\mathsf{F}\neg A\vee\mathsf{F}\neg B)\vee\mathsf{G}(A\wedge B))\wedge(\mathsf{F}(\neg A\vee\neg B)\vee(\mathsf{G}A\wedge\mathsf{G}B))}}(\wedge)$$

The main formula of each step of the proof is printed in bold face. Clearly, this proof is much more goal oriented and looks much better to automate than the Hilbert proof of the previous chapter.

10.5 Properties

In this chapter we look at some properties of the Tait Calculus. We already know that the calculus is sound and complete, i.e. that we can do the same in this calculus as in the original Hilbert calculus and thus with the semantics of LL$^-$.

Subformula Property

Definition 22 ***Subformulas*** subfml($\cdot$)

If A is a formula, then the set of subformulas subfml(A) *is inductively defined as follows:*

$$\begin{array}{ll}
A \equiv p_i & \textit{then } \mathrm{subfml}(A) := \{p_i\}\ , \\
A \equiv \neg B & \textit{then } \mathrm{subfml}(A) := \mathrm{subfml}(B) \cup \{\neg B\}, \\
A \equiv B \vee C & \textit{then } \mathrm{subfml}(A) := \mathrm{subfml}(B) \cup \mathrm{subfml}(C) \cup \{B \vee C\}, \\
A \equiv B \wedge C & \textit{then } \mathrm{subfml}(A) := \mathrm{subfml}(A) \cup \mathrm{subfml}(B) \cup \{B \wedge C\}, \\
A \equiv \mathsf{L}B & \textit{then } \mathrm{subfml}(A) := \mathrm{subfml}(B) \cup \{\mathsf{L}B\}, \\
A \equiv \mathsf{K}B & \textit{then } \mathrm{subfml}(A) := \mathrm{subfml}(B) \cup \{\mathsf{K}B\}, \\
A \equiv \mathsf{F}B & \textit{then } \mathrm{subfml}(A) := \mathrm{subfml}(B) \cup \{\mathsf{F}B\}, \\
A \equiv \mathsf{G}B & \textit{then } \mathrm{subfml}(A) := \mathrm{subfml}(B) \cup \{\mathsf{G}B\}.
\end{array}$$

Lemma 4 ***Number of Subformulas*** $|\mathrm{subfml}(A)|$

If A is a formula then

$$|\mathrm{subfml}(A)| \leq |A|$$

Proof

This can easily been shown by induction over the structure of A.

Definition 23 ***Subformula Property***

If Γ is the conclusion of a rule and Δ one of it's premisses, then the rule has the subformula property iff

$$B \in \Delta \quad \Rightarrow \quad B \in \mathrm{subfml}(A) \textit{ for a } A \in \Gamma$$

Lemma 5 ***Subformula Property***

Every rule in the Tait calculus has the subformula property.

Proof

A closer look at the individual rules of the calculus shows that every formula of a premiss either already exists in the conclusion or is a subformula of one of these formulas.

It is important to note here that not all rules of the Hilbert calculus presented in the previous chapter have the subformula property. The modus ponens rule used for that calculus introduces a new formula in the premiss. This creates serious problems for implementing the calculus in an automated theorem prover, as mentioned in the previous chapter.

10.5.1 Contraction

Contraction is the fact that from $\models^n_T A, A, \Gamma$ follows $\models^n_T A, \Gamma$. Because we are dealing with sets in the Tait calculus, the set A, A, Γ is actually the same as the set A, Γ.

Thus, the Tait calculus surely has contraction. We just mention it here, because the same proposition will be examined for the later calculi, where we deal with sequents of formulas instead of sets. In these cases, this property is no more trivial.

10.5.2 Invertible Rules

Definition 24 ***Invertible Rule***

A rule $\dfrac{\Delta_0 \quad \Delta_1 \quad \ldots \quad \Delta_i}{\Gamma}$ *of the Tait calculus is* invertible *iff*

$$\models_T \Gamma \quad \Rightarrow \quad (\models_T \Delta_j \textit{ for all } 0 \leq j \leq i)$$

A rule is invertible, if each premiss is provable if the conclusion is provable.

Definition 25 ***Strongly Invertible Rule***

A rule $\dfrac{\Delta_0 \quad \Delta_1 \quad \ldots \quad \Delta_i}{\Gamma}$ *of the Tait calculus is* strongly invertible, *iff for all* $n \in \mathbb{N}$

$$\models^n_T \Gamma \quad \Rightarrow \quad (\models^n_T \Delta_j \textit{ for all } 0 \leq j \leq i)$$

A strongly invertible rule is one that is invertible and whose premisses can be proven with the same length as its conclusion.

Lemma 6 ***Strongly Invertible Rules of the Tait Calculus***
The rules ($\wedge$), ($\vee$) are strongly invertible.

As an example we prove the strong invertibility of the ($\vee$)-rule:

Proof
We assume that the conclusion has the form $A \vee B, \Gamma$.

We make an induction over the structure of the proof of the Conclusion:

- if $A \vee B$ was the main formula of the last step—which thus must have been a $(\vee)$-rule—we get the assumption directly from the premiss of the rule and the induction hypothesis[5], with the same proof length.
- if the last rule used was (id), then $A \vee B, \Gamma$ must be an axiom. Because axioms must contain a propositional variable and its negation, Γ itself must be an axiom ($A \vee B$ cannot be a propositional variable). Thus, the same proof can be done for the premiss, using the same length
- if the last step was a $(\vee)$, (L), or a (F)-rule, we can use the induction hypothesis on its premiss and carry through the same step to get the desired proof, with the same length.
- if the last step was a $(\wedge)$-rule, then we can use the induction hypothesis on both premisses and carry through the same step to get the proof we are looking for, with the correct length.
- if the last step was a (G) or (K)-rule, then the formula $A \vee B$ cannot appear in the premiss, because it does not start with either a L or F. Thus, from the same premiss using the rule's builtin weakening, we get the proof we are looking for, with the same length as before.

The proof for the $(\wedge)$-rule is almost the same.

Although the (L) and (F)-rules don't seem to be invertible at first glance, they are nevertheless invertible in a special case. Because we deal with sets in a Tait calculus, the formula $\mathsf{L}A$ may appear on the premiss of the rule as well (i.e. $\{\mathsf{L}A\} \cap \Gamma = \{\mathsf{L}A\}$). In that case the rule is strongly invertible, otherwise it is not.

Example 10 ***Invertibility***
As an example we set Γ to the empty set. In this case, invertibility would for example mean $\models_T \mathsf{L}p_i, \neg\mathsf{L}p_i \;\Rightarrow\; \models_T p_i\neg, \mathsf{L}p_i$. While the former is always provable, the latter surely is not.[6]

It has to be noted that the (K) and (G)-rules are *not* invertible. As a simple example consider the following set of formulas

$$p_{\mathrm{i}}, \neg p_{\mathrm{i}}, GB.$$

This is surely an axiom and thus provable. But the premiss of the (G)-rule is just B and in general not provable.

10.5.3 Backward Proof Search

A formula A is provable in the Tait calculus if and only if there exists a derivation sequence which starts with axioms, ends with the formula A and only uses the rules of the calculus. When doing

[5] for the case that $A \vee B$ appears in the premiss as well

[6] the latter is actually $\mathsf{L}p_i \to p_i$, and thus surely not provable.

backward proof search to actually find a proof of a formula, we do the whole process backwards. We start with the formula to prove and apply rules backwards until we arrive at axioms in each branch or don't have any rule to go on with. A tree built in such a way that ends in axioms only is clearly a proof of the formula.

Example 11 ***Backward Proof***

Clearly, not every backward proof search ends in axioms only, even if the formula is provable. Consider the following example of the proof of $\mathsf{F}\neg B \vee \mathsf{G}A \vee \mathsf{G}B$ (which is actually $\neg\mathsf{G}B \vee \mathsf{G}A \vee \mathsf{G}B$ in negation normal form):

$$
\begin{prooftree}
\AxiomC{$\neg B, A$}
\RightLabel{F}
\UnaryInfC{$\mathsf{F}\neg B, A$}
\RightLabel{G}
\UnaryInfC{$\mathsf{F}\neg B, \mathsf{G}A, \mathsf{G}B$}
\RightLabel{$\vee$}
\UnaryInfC{$\mathsf{F}\neg B, \mathsf{G}A \vee \mathsf{G}B$}
\RightLabel{$\vee$}
\UnaryInfC{$\mathsf{F}\neg B \vee \mathsf{G}A \vee \mathsf{G}B$}
\end{prooftree}
$$

This proof does not work, i.e. we come to a place where we can't go on. The problem lies in the fact that the (G)-rule used the wrong main formula. If we use $\mathsf{G}B$ instead of $\mathsf{G}A$ as the main formula we get a successful proof:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{id}
\UnaryInfC{$\neg B, B$}
\RightLabel{F}
\UnaryInfC{$\mathsf{F}\neg B, B$}
\RightLabel{G}
\UnaryInfC{$\mathsf{F}\neg B, \mathsf{G}A, \mathsf{G}B$}
\RightLabel{$\vee$}
\UnaryInfC{$\mathsf{F}\neg B, \mathsf{G}A \vee \mathsf{G}B$}
\RightLabel{$\vee$}
\UnaryInfC{$\mathsf{F}\neg B \vee \mathsf{G}A \vee \mathsf{G}B$}
\end{prooftree}
$$

This example shows that the main formula selected for a rule, as well as the general sequence of the rules determines if a proof will be successful or not. Thus, when trying to find a proof of a formula or to know that the formula is not provable, all possible selections and sequences of rule applications have to be tried. This might require backtracking, if the selected formula did, for example, not end in a successful proof.

For backward proof searches, invertible rules have big advantages. With an invertible rule, the premiss is provable iff the conclusion is provable, thus they never require backtracking. If a proof can be made starting with the conclusion of a rule, a proof can also be made starting with its premiss. Thus, invertible rules can always be applied, without fear of dead ends. This can speed up proof search considerably.

10.5.4 Termination

In general, the given Tait calculus does not terminate. Because we are dealing with sets, for all invertible rules the main formula can also appear on the premiss of the rule. But if the main formula of a rule also appears in the premiss, this clearly may result in an infinite loop. Thus, backward proof search would not terminate.

To be able to include the main formula of a rule in the premiss is actually essential for the (L)- and (F)-rules to get completeness of the calculus. Unfortunately, this prohibits an efficient implementation of the calculus in an automated theorem prover, because now the rule is not invertible. Furthermore, the theorem prover does not always terminate.

10.5.5 Disjunction Principle

Lemma 7 ***Disjunction Principle***

$$\begin{array}{lcl} \vdash_{T} \mathsf{G}A_1, \ldots, \mathsf{G}A_n, \mathsf{K}B_1, \ldots, \mathsf{K}B_n & \Rightarrow & \vdash_{T} \mathsf{G}A_1 \quad \textit{or} \quad \ldots \quad \textit{or} \quad \vdash_{T} \mathsf{G}A_n \quad \textit{or} \\ & & \vdash_{T} \mathsf{K}B_1 \quad \textit{or} \quad \ldots \quad \textit{or} \quad \vdash_{T} \mathsf{K}B_m \end{array}$$

Proof
Let $\vdash_{T} \mathsf{G}A_1, \ldots, \mathsf{G}A_n, \mathsf{K}B_1, \ldots, \mathsf{K}B_n$.

Surly this can't be an axiom, because we only allow propositional variables in axioms. Thus, there has to be a last step of the proof. We distinguish two different cases:

- the main formula of the last step was $\mathsf{G}A_i$.

  In this case, the premiss of the rule is just A_i. By using the same rule again, but without weakening, we get the desired proof of $\mathsf{G}A_i$.

- the main formula of the last step was $\mathsf{K}B_i$.

  In this case, the premiss of the rule is just B_i. Again, by using the same rule again, but without weakening, we get the desired proof of $\mathsf{K}B_i$.

The disjunction principle says that if we have to prove two G-formulas, we can make a proof using one or the other, but we will never have to use both formulas.

This makes backward proof search much easier and also much faster. Each time a G-formula is considered as main formula, all other G-formulas can be removed. Furthermore, the sequence which is used to determine which G-formula to select as main formula does not influence provability. On the other hand, it also requires that the application of (G) and (K) rules is made last in the proof search, or this principle cannot be used.[7]

10.6 Deduction Theorem

For the practical use of the logic of likelihood, it is often essential to add special, non-logical axioms to the proof search. These axioms are assumed to be valid and thus can directly be used in a proof. Contrary to the logical axioms of the Hilbert calculus, substitution of variables for arbitrary formulas is not allowed with these types of axioms. The only assumption made is that non-logical axioms are assumed to be true.

Definition 26 ***Theories***
We say a set Δ is derivable in the Tait calculus with cut from a finite theory Σ, *written as* $\Sigma \vdash_{T+C} \Delta$ *if there is a proof of Δ using only rules of the Tait calculus and the (cut)-rule and additionally treating all elements of Σ as extra-logical axioms, i.e.*

$$\frac{}{A, \Gamma}\ (\Sigma)$$

is an axiom for all $A \in \Sigma$.

Using theories with a calculus is only useful when also allowing the (cut)-rule.[8] Therefore provability using theories is only defined for the Tait calculus with (cut).

Theorem 6 ***Deduction Theorem***

$$\Sigma \vdash_{T+C} \Delta \quad \Leftrightarrow \quad \vdash_{T+C} \mathsf{F}\neg\Sigma, \Delta$$

Proof

$\Leftarrow$: We assume $\vdash_{T+C} \mathsf{F}\neg\Sigma, \Delta$. Furthermore, we know that $\Sigma \vdash_{T+C} B_i$ for all $B_i \in \Sigma$. Using generalization this is $\Sigma \vdash_{T+C} \mathsf{G}B_i$.

Because $\neg\mathsf{G}B_i \equiv \mathsf{F}\neg B_i$, we can use $|\Sigma|$ cuts to get to $\Sigma \vdash_{T+C} \Delta$, as desired.

$\Rightarrow$: This time we assume $\Sigma \vdash_{T+C} \Delta$. Now we make an induction on the length n of the proof of Δ using the theory Σ.

$n = 0$: in that case, Δ is an axiom.

If $\Delta \cap \Sigma = \emptyset$, then Δ is an axiom of the Tait calculus with cut and thus $\vdash_{T+C} \Delta$

Otherwise, there is a formula $A \in \Delta$ with $A \in \Sigma$. Thus we can use a (F)-rule to get from $\mathsf{F}\neg A, A, \Delta'$ to $\neg A, A, \Delta'$. According to lemma 3 this can be proven in the Tait calculus, thus $\vdash_{T+C} \mathsf{F}\neg A, \Delta$ surely holds.

[7] the disjunction principle is not provable for sets containing arbitrary formulas.

[8] it's not possible to deduce A from the theory $A \wedge B$ otherwise.

$n > 0$: if Δ still is an axiom, we proceed as in $n = 0$. Thus we assume that the last step of the proof was a rule application and that the main formula of this rule was A. We now make a distinction between the last rule used:

$\vee$: $A \equiv B \vee C$
From the last step of the rule we know $\Sigma \vdash_{T+C} B, C, \Delta'$. Using the induction hypothesis we get $\vdash_{T+C} \mathsf{F}\neg\Sigma, B, C, \Delta'$. Using the ($\vee$)-rule again yields the desired proof.

$\wedge$: $A \equiv B \wedge C$
From the last step of the rule we know $\Sigma \vdash_{T+C} B, \Delta'$ and $\Sigma \vdash_{T+C} C, \Delta'$. Using the induction hypothesis twice we get $\vdash_{T+C} \mathsf{F}\neg\Sigma, B, \Delta'$ and $\vdash_{T+C} \mathsf{F}\neg\Sigma, C, \Delta'$. Using the ($\wedge$)-rule again yields the desired proof.

L: $A \equiv \mathsf{L}B$
From the last step of the rule we know $\Sigma \vdash_{T+C} B, \Delta'$. Using the induction hypothesis we get $\vdash_{T+C} \mathsf{F}\neg\Sigma, B, \Delta'$. Using the ($\mathsf{L}$)-rule again yields the desired proof.

K: $A \equiv \mathsf{K}B$
From the last step of the rule we know $\Sigma \vdash_{T+C} B, \Delta'_{\mathsf{L}}, \mathsf{F}\Delta'_{\mathsf{F}}$. Using the induction hypothesis we get $\vdash_{T+C} \mathsf{F}\neg\Sigma, B, \Delta'_{\mathsf{L}}, \mathsf{F}\Delta'_{\mathsf{F}}$. Using the ($\mathsf{F}$)-rule again yields the desired proof, because *all* F-formulas are taken from the premiss to the conclusion, including the newly introduced ones.

F: $A \equiv \mathsf{F}B$
From the last step of the rule we know $\Sigma \vdash_{T+C} B, \Delta'$. Using the induction hypothesis we get $\vdash_{T+C} \mathsf{F}\neg\Sigma, B, \Delta'$. Using the ($\mathsf{F}$)-rule again yields the desired proof.

G: $A \equiv \mathsf{G}B$
From the last step of the rule we know $\Sigma \vdash_{T+C} B, \mathsf{F}\Delta'_{\mathsf{F}}$. Using the induction hypothesis we get $\vdash_{T+C} \mathsf{F}\neg\Sigma, B, \mathsf{F}\Delta'_{\mathsf{F}}$. Using the ($\mathsf{F}$)-rule again yields the desired proof, because all F-formulas are taken from the premiss to the conclusion.

cut: From the last step of the rule we know $\Sigma \vdash_{T+C} A, \Delta'_1$ and $\Sigma \vdash_{T+C} \neg A, \Delta'_2$. By using the induction hypothesis twice we get proofs for $\vdash_{T+C} \mathsf{F}\neg\Sigma, A, \Delta'_1$ and $\vdash_{T+C} \mathsf{F}\neg\Sigma, \neg A, \Delta'_2$. Using the (cut)-rule again yields the desired proof.

🐧

The G on the right side of the deduction theorem is necessary. It is not true that

$$\Sigma \vdash_{T+C} A \quad \Rightarrow \quad \vdash_{T+C} \neg\Sigma, A.$$

This can easily be seen with the next example or by examining the parts for the (G) and (K)-rules in the proof of the deduction theorem. In order to be able to use the induction hypothesis an the premiss of the rule and then still be able to apply the rule again, all formulas in the premiss are required to start with an F-operator, or the rule cannot be applied.

Example 12 ***Deduction Theorem***
As an example, we consider the case that $\Sigma = \{B\}$ and $A \equiv \mathsf{G}B$. Then, with generalization, we can surely prove $\mathsf{G}B$ from B.

On the other, $\neg B, \mathsf{G}B$ is surely not provable, because this is the same as $B \to \mathsf{G}B$. As a counter model, we can take a simple model with two states, where in the first state B is true and in the second, $\mathscr{C}$-reachable state is not.

The deduction theorem is very important for automated backward proof search with theories. Adding theory support to an algorithm for automated theorem proving only requires to transform the theory along with the formula to prove. This new formula can then be proven normally. Thus, the proving algorithm only requires some preprocessing steps, but no changes in the algorithm itself.

The following corollary allows us to easily use theories with our Tait calculus without cut.

Corollary 1 ***Deduction Theorem***

$$\Sigma \vdash_{T+C} A \quad \Leftrightarrow \quad \vdash_{T} (\bigwedge \mathsf{G}\Sigma) \to A$$

Proof
First we use the deduction theorem to get

$$\Sigma \vdash_{T+C} A \quad \Leftrightarrow \quad \mathsf{F}\neg\Sigma, A$$

Then we use lemma 2 to get

$$\Sigma \vdash_{T+C} A \quad \Leftrightarrow \quad \vdash_{T} \mathsf{F}\neg\Sigma, A$$

As a last step we use the fact that $\neg\mathsf{G}A \equiv \mathsf{F}\neg A$ and that $\mathsf{G}(A \wedge B) \leftrightarrow (\mathsf{G}A \wedge \mathsf{G}B)$ to get the desired result.

This version of the deduction theorem allows us to use theories for our Tait calculus without cut, thus allowing easy implementation of theory handling in an automated theorem prover.

This version also adds a simple transformation to the theory. Using this in a theorem prover can improve its performance, especially for large theories. Now, the ($\wedge$) rule can be applied before having to use the (G)-rule. Because the ($\wedge$)-rule is invertible, it never requires backtracking. The (G)-rule, on the other hand, always forces backtracking if more than one G-formula is present.

Chapter 11

Sequent Calculi

This chapter defines another calculus to be used for proof search. As mentioned in the previous chapter, the Tait calculus has some serious drawbacks for implementations. Mainly, it does not necessarily terminate.

11.1 Notation

For this chapter we use the following, slightly different notation:

- We use the symbols Γ, Δ, Π, and Σ for multisets of formulas, i.e. the same formula may now appear multiple times.
- Γ' denotes sub-multisets of Γ; the particular multiset will be determined by its context
- Γ_{L} stands for a multiset consisting of some L-formulas[1] of Γ, with the leading L removed; it is *not* necessary that Γ_{L} consist of *all* L-formulas of Γ. Γ_{F}, Γ_{K} and Γ_{G} are similarly defined.
- $\bigvee \Gamma$ stands for the disjunction of all elements of Γ. Similarly, $\bigwedge \Gamma$ stands for the conjunction of all elements of Γ.
- nnf(A) is the negation normal form of A, and nnf(Γ) is the negation normal form of all formulas of the multiset Γ.
- $\neg\Gamma$ stands for the multiset of formulas in which every individual formula of Γ has been negated.
- We write $\Gamma \cup \{A\}$, or sometimes $\Gamma \cup A$ for the multiset Γ extended by adding the formula A.

Example 13 ***Notations***
We look at several examples to show the meaning of these notations. For all these examples we assume that $\Gamma := p_1, \mathsf{F}p_2, \mathsf{L}p_3, \mathsf{F}(\neg(p_4 \vee p_2)), p_1, \neg\mathsf{G}p_1, \mathsf{G}p_2, \mathsf{LF}p_2$.

- Γ_{L} could be $\Gamma_{\mathsf{L}} = p_3, \mathsf{F}p_2$, but it could also be just p_3 or $\mathsf{F}p_2$ or even the empty set.

[1] formulas starting with L

- Γ_{F} could be $\Gamma_{\mathsf{F}} = \neg(p_4 \vee p_2)$, but it could also be $p_2, \neg(p_4 \vee p_2)$, just p_2, or again even the empty set.
- $\bigvee \Gamma = p_1 \vee \mathsf{F}p_2 \vee \mathsf{L}p_3 \vee \mathsf{F}(\neg(p_4 \vee p_2)) \vee p_1 \vee \neg\mathsf{G}p_1 \vee \mathsf{G}p_2 \vee \mathsf{LF}p_2$.
- $\bigwedge \Gamma = p_1 \wedge \mathsf{F}p_2 \wedge \mathsf{L}p_3 \wedge \mathsf{F}(\wedge(p_4 \vee p_2)) \wedge p_1 \wedge \neg\mathsf{G}p_1 \wedge \mathsf{G}p_2 \wedge \mathsf{LF}p_2$.
- nnf(()$\Gamma) = p_1, \mathsf{F}p_2, \mathsf{L}p_3, \mathsf{F}(\neg p_4 \wedge \neg p_2)), p_1, \neg\mathsf{G}p_1, \mathsf{G}p_2, \mathsf{LF}p_2$.
- $\neg\Gamma = \neg p_1, \neg\mathsf{F}p_2, \neg\mathsf{L}p_3, \neg\mathsf{F}(\neg(p_4 \vee p_2)), \neg p_1, \neg\neg\mathsf{G}p_1, \neg\mathsf{G}p_2, \neg\mathsf{LF}p_2$.

11.2 Sequents

Throughout this chapter, we will use several forms of sequents of formulas for the calculi. Sequents for sequent calculi are primarily split into two main patterns, one-sided and two-sided sequents, giving one-sided and two-sided sequent calculi. In this chapter we will only use double-sided sequents, because such sequents are better suited for implementation than one-sided sequents. Using double-sided sequents allows algorithms to directly deal with implications and equivalences.

A double sided sequents basically consist of two multisets delimited by the '$\supset$' sign. Sequents of the form $\Gamma \supset \Delta$ can be interpreted as meaning $\bigwedge \Gamma \to \bigvee \Delta$, i.e. the delimiter is treated as an implication.

The sequent calculus introduced below requires more complicated sequents. These sequents consists of the basic two sequents, but each of the two sequent sides gets two additional multisets containing specially marked formulas. The left side additionally has multisets for marked K and G-formulas, while the right side has two additional multisets for marked L and F-formulas. These additional multisets are used to remember formulas already treated, to prevent simple loops.

11.3 Double-Sided Sequent Calculus

The next page show a double-sided sequent calculus that can be used to prove formulas of LL^-. The main reason to even introduce an additional calculus are the serious drawbacks of the Tait and Hilbert style calculi mention in earlier chapters. These drawbacks make implementation of the calculi in an automated theorem prover much harder, a situation which is improved by the following double-sided sequent calculus.

After the calculus, we will first look at some motivation why this particular calculus was selected. Then follows the necessary correctness and completeness proofs for the calculus. Afterwards, we look at some properties of the calculus. This will include some remarks about implementing it with an automated theorem prover. Because of some implementation problems, we will then have to adjust the calculus to finally obtain the calculus that was actually implemented in the Logics Workbench.

$$\frac{}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid p_i,\Gamma \supset p_i,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(id)} \qquad \frac{}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{false},\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(id-f)} \qquad \frac{}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \mathsf{true},\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(id-t)}$$

$$\frac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset A,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \neg A,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(l-}\neg\text{)} \qquad \frac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid A,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \neg A,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(r-}\neg\text{)}$$

$$\frac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid A,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \quad \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid A \vee B,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(l-}\vee\text{)} \qquad \frac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset A,B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset A \vee B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(r-}\vee\text{)}$$

$$\frac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid A,B,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid A \wedge B,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(l-}\wedge\text{)} \qquad \frac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset A,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \quad \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset A \wedge B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(r-}\wedge\text{)}$$

$$\frac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset A,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \quad \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid A \rightarrow B,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(l-}\longrightarrow\text{)} \qquad \frac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid A,\Gamma \supset B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset A \rightarrow B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(r-}\longrightarrow\text{)}$$

$$\frac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset A,B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \quad \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid A,B,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid A \leftrightarrow B,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(l-}\longleftrightarrow\text{)} \qquad \frac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid A,\Gamma \supset B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \quad \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B,\Gamma \supset A,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset A \leftrightarrow B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(r-}\longleftrightarrow\text{)}$$

$$\frac{\emptyset;\Pi_{\mathsf{G}} \mid A,\Pi_{\mathsf{K}},\Pi_{\mathsf{G}} \supset \Sigma_{\mathsf{L}},\Sigma_{\mathsf{F}} \mid \emptyset;\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{L}A,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(l-L)} \qquad \frac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset A,\Delta \mid \Sigma_{\mathsf{L}} \cup A;\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \mathsf{L}A,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(r-L)}$$

$$\frac{\emptyset;\Pi_{\mathsf{G}} \mid A,\Pi_{\mathsf{G}} \supset \Sigma_{\mathsf{F}} \mid \emptyset;\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{F}A,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(l-F)} \qquad \frac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset A,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \cup A}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \mathsf{F}A,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(r-F)}$$

$$\frac{\Pi_{\mathsf{K}} \cup A;\Pi_{\mathsf{G}} \mid A,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{K}A,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(l-K)} \qquad \frac{\emptyset;\Pi_{\mathsf{G}} \mid \Pi_{\mathsf{K}},\Pi_{\mathsf{G}} \supset A,\Sigma_{\mathsf{L}},\Sigma_{\mathsf{F}} \mid \emptyset;\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \mathsf{K}A,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(r-K)}$$

$$\frac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \cup A \mid A,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{G}A,\Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(l-G)} \qquad \frac{\emptyset;\Pi_{\mathsf{G}} \mid \Pi_{\mathsf{G}} \supset A,\Sigma_{\mathsf{F}} \mid \emptyset;\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \mathsf{G}A,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\ \text{(r-G)}$$

Definition 27 ***Rule Names***
We call the rules (l-$\neg$), (r-$\neg$), (l-$\vee$), (r-$\vee$), (l-$\wedge$), (r-$\wedge$), (l-$\longrightarrow$), (r-$\longrightarrow$), (l-$\leftrightarrow$), (r-$\leftrightarrow$) the classical *rules of the sequent calculus.*

*The rules (r-*L*), (r-*F*), (l-*K*), and (l-*G*) we call* L/F *rules.*

*The remaining rules (l-*L*), (l-*F*), (r-*K*), and (r-*G*) we call* K/G *rules.*

The classical rules are those that are the same as for a classical sequent calculus. The L/F-rules actually are just marking rules, where L and F-formulas are marked for later use. The K/G-rules are the most complicated rules. They may require backtracking when doing backward proof search.

11.4 Motivation

The axioms of the calculus are the same as the one of the Tait calculus. We just treat the standard sequents here and also added axioms to directly deal with the constants true and false.

The classical sequents are also similar to the one of the Tait calculus. Because we are no more just dealing with formulas in negation normal form, we have to treat negation, implication and equivalence as well. Furthermore, these rules have to deal with the whole sequents used for the other rules.

The rules (r-L), (r-F), (l-K), and (l-G) (L/F-rules) are the same as the ones of the Tait calculus. Instead of dealing with sets, we now have multisets. Furthermore, to prevent loops by taking the same formula as main formula of the rule again and again, we mark the formula used by putting them in a special multi set. Marked formulas can no more be used for this rule.

The rule (r-K), (r-G), (l-L), and (l-F) (K/G-rules) are similar to the Tait rules, but have to take care of the marked formulas as well. Such formulas are unmarked and again available to be used for application by other rules. This is necessary, as the next example shows.

Example 14 ***Unmarking Formulas***
The formula $\mathsf{G}p \longrightarrow \mathsf{GG}p$, written in negation normal form as $\mathsf{F}\neg p \vee \mathsf{GG}p$ should surely be provable in our sequent calculus. The following proof shows that it is provable in our calculus.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset;\emptyset \mid p \supset p \mid \emptyset;\neg p}\,(\mathrm{id})}{\emptyset;\emptyset \mid \quad \supset p, \neg p \mid \emptyset;\neg p}\,(\neg)}{\emptyset;\emptyset \mid \quad \supset \mathsf{G}p, \neg p \mid \emptyset;\neg p}\,(\mathsf{G})}{\emptyset;\emptyset \mid \quad \supset \mathsf{GG}p \mid \emptyset;\neg p}\,(\mathsf{G})}{\emptyset;\emptyset \mid \quad \supset \mathsf{F}\neg p, \mathsf{GG}p \mid \emptyset;\emptyset}\,(\mathsf{F})}{\emptyset;\emptyset \mid \quad \supset \mathsf{F}\neg p \vee \mathsf{GG}p \mid \emptyset;\emptyset}\,(\vee)$$

This example shows that it is necessary to put back the marked formulas into the sides of the sequent. Additionally, it is also necessary to keep these same formulas marked, to take care of the transitivity of G.

11.5 Correctness

We prove correctness of the sequent calculus by transforming a proof of the sequent calculus into one of the Tait calculus of the previous chapter. Because the Tait calculus only dealt with formulas in negation normal form, we have to do this transformation here as well.

Theorem 7 ***Correctness of the Sequent Calculus***
$\vdash_S \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \quad \rightarrow \quad \vdash_T \mathrm{nnf}(\neg\mathsf{K}\Pi_{\mathsf{K}}, \neg\mathsf{G}\Pi_{\mathsf{G}}, \neg\Gamma, \Delta, \mathsf{L}\Sigma_{\mathsf{L}}, \mathsf{F}\Sigma_{\mathsf{F}})$

Proof
We make the proof by induction on the length n of the proof:

$n = 0$ We have one of the axioms (id), (id-t), or (id-f)

If we have the axiom (id), then our assumption to prove is

$$\mathrm{nnf}(\neg\mathsf{K}\Pi_{\mathsf{K}}, \neg\mathsf{G}\Pi_{\mathsf{G}}, \neg A, \neg\Gamma', A, \Delta', \mathsf{L}\Sigma_{\mathsf{L}}, \mathsf{F}\Sigma_{\mathsf{F}}).$$

While this is not necessarily directly an axiom of the Tait calculus, it is easy to see that this formula is provable there, because it is a classical tautology.

If we have the axiom (id-t), then with $\mathsf{true} \equiv p_i \vee \neg p_i$ we can make the following proof of our assumption in the Tait calculus:

$$\dfrac{\dfrac{}{\neg\mathsf{K}\Pi_{\mathsf{K}}, \neg\mathsf{G}\Pi_{\mathsf{G}}, \neg\Gamma, p_i, \neg p_i, \Delta', \mathsf{L}\Sigma_{\mathsf{L}}, \mathsf{F}\Sigma_{\mathsf{F}}}\,(\mathrm{id})}{\neg\mathsf{K}\Pi_{\mathsf{K}}, \neg\mathsf{G}\Pi_{\mathsf{G}}, \neg\Gamma, p_i \vee \neg p_i, \Delta', \mathsf{L}\Sigma_{\mathsf{L}}, \mathsf{F}\Sigma_{\mathsf{F}}}\,(\vee)$$

If we have the axiom (id-f), then with $\mathsf{false} \equiv p_i \wedge \neg p_i$ and with $\mathrm{nnf}(\neg(p_i \wedge \neg p_i) \equiv \neg p_i \vee p_i$ and the proof above follows the assumption.

$n > 0$ If we still only used one of the axioms in the proof, then we argue as in the case of $n = 0$.

Otherwise, we used one of the other rules. Thus, we check all the remaining rules individually:

(l-$\neg$) the induction hypothesis gives us

$$\vdash_T \mathrm{nnf}(\neg\mathsf{K}\Pi_{\mathsf{K}}, \neg\mathsf{G}\Pi_{\mathsf{G}}, \neg A, \neg\Gamma, \Delta, \mathsf{L}\Sigma_{\mathsf{L}}, \mathsf{F}\Sigma_{\mathsf{F}}).$$

But this is the same as the conclusion, thus the assumption holds.

(r-$\neg$) this is the same as the case (l-$\neg$).

(l-$\vee$) again, from the induction hypothesis we get that the premisses of the (l-$\vee$) are provable in the Tait calculus. Thus

$$\vdash_{\mathcal{T}} \text{nnf}(\neg \mathsf{K}\Pi_{\mathsf{K}}, \neg \mathsf{G}\Pi_{\mathsf{G}}, \neg A, \neg\Gamma, \Delta, \mathsf{L}\Sigma_{\mathsf{L}}, \mathsf{F}\Sigma_{\mathsf{F}})$$

and

$$\vdash_{\mathcal{T}} \text{nnf}(\neg \mathsf{K}\Pi_{\mathsf{K}}, \neg \mathsf{G}\Pi_{\mathsf{G}}, \neg B, \neg\Gamma, \Delta, \mathsf{L}\Sigma_{\mathsf{L}}, \mathsf{F}\Sigma_{\mathsf{F}})$$

With the application of a ($\wedge$)-rule and the fact that nnf($\neg(A \vee B)$) $=$ nnf($\neg A$) $\wedge$ nnf($\neg B$) follows the assumption.

(r-$\vee$) from the induction hypothesis we know that

$$\vdash_{\mathcal{T}} \text{nnf}(\neg \mathsf{K}\Pi_{\mathsf{K}}, \neg \mathsf{G}\Pi_{\mathsf{G}}, \neg\Gamma, A, B, \Delta, \mathsf{L}\Sigma_{\mathsf{L}}, \mathsf{F}\Sigma_{\mathsf{F}}).$$

With the application of a ($\vee$)-rule follows the assumption.

(l-$\wedge$) this case is similar to the case (r-$\vee$).

(r-$\wedge$) this case is similar to the case (l-$\vee$).

(l-$\rightarrow$) The assumption follows directly from the induction hypothesis and the application of a ($\wedge$) rule with the fact that nnf($\neg(A \rightarrow B)$) $\equiv A \wedge \neg B$.

(r-$\rightarrow$) The assumption follows directly from the induction hypothesis and the application of a ($\vee$) rule with the fact that nnf($A \rightarrow B$) $\equiv A \vee \neg B$.

(l-$\leftrightarrow$) The assumption follows directly from the induction hypothesis and the application of two ($\vee$)-rules and a ($\wedge$)-rule and the fact that nnf($\neg(A \leftrightarrow B)$) $\equiv (A \vee B) \wedge (\neg A \vee \neg B)$.

(r-$\leftrightarrow$) The assumption follows directly from the induction hypothesis and the application of two ($\vee$)-rules and a ($\wedge$)-rule and the fact that nnf($A \leftrightarrow B$) $\equiv (\neg A \vee B) \wedge (A \vee \neg B)$.

(l-L) From the premiss of the rule and the induction hypothesis we immediately get

$$\vdash_{\mathcal{T}} \text{nnf}(\neg \mathsf{G}\Pi_{\mathsf{G}}, \neg A, \neg\Pi_{\mathsf{K}}, \neg\Pi_{\mathsf{G}}, \Sigma_{\mathsf{L}}, \Sigma_{\mathsf{F}}, \mathsf{F}\Sigma_{\mathsf{F}}).$$

By using the fact that nnf($\neg \mathsf{G}B$) $= \mathsf{F}$ nnf($\neg B$) this is

$$\vdash_{\mathcal{T}} \text{nnf}(\mathsf{F}\neg\Pi_{\mathsf{G}}, \neg A, \neg\Pi_{\mathsf{K}}, \neg\Pi_{\mathsf{G}}, \Sigma_{\mathsf{L}}, \Sigma_{\mathsf{F}}, \mathsf{F}\Sigma_{\mathsf{F}}).$$

We can now apply a (F)-rule for every member of $\neg\Pi_{\mathsf{G}}$ and Σ_{F} to get[2]

$$\vdash_{\mathcal{T}} \text{nnf}(\mathsf{F}\neg\Pi_{\mathsf{G}}, \neg A, \neg\Pi_{\mathsf{K}}, \Sigma_{\mathsf{L}}, \mathsf{F}\Sigma_{\mathsf{F}}).$$

Using the appropriate weakenings, this can be used as the premiss of the (K)-rule to get,

$$\vdash_{\mathcal{T}} \text{nnf}(\neg \mathsf{K}\Pi_{\mathsf{K}}, \mathsf{F}\neg\Pi_{\mathsf{G}}, \mathsf{K}\neg A, \neg\Gamma, \Delta, \mathsf{L}\Sigma_{\mathsf{L}}, \mathsf{F}\Sigma_{\mathsf{F}}).$$

Now, again with some properties of the negation normal form this is

$$\vdash_{\mathcal{T}} \text{nnf}(\mathsf{L}\neg\Pi_{\mathsf{K}}, \mathsf{F}\neg\Pi_{\mathsf{G}}, \neg \mathsf{L}A, \Gamma, \Delta, \mathsf{L}\Sigma_{\mathsf{L}}, \mathsf{F}\Sigma_{\mathsf{F}}),$$

which is the desired assumption.

[2] we left out the double appearance of $\mathsf{F}\neg\Pi_{\mathsf{G}}$ and $\mathsf{F}\Sigma_{\mathsf{F}}$ to make things a bit easier; these formulas would not disturb the further proof steps in any way.

(r-L) As before, the assumption follows from the induction hypothesis and the application of a (K)-rule, weakening and the fact that $\text{nnf}(\neg\mathsf{L}A) \equiv \mathsf{K}\neg\text{nnf}(A)$ and $\text{nnf}(\neg\mathsf{F}A) \equiv \mathsf{G}\neg\text{nnf}(A)$.

(l-F) this case is similar to the case of (l-L).

(r-F) this case is similar to the case of (r-L).

(l-K) this case is similar to the case of (r-L).

(r-K) this case is similar to the case of (l-L).

(l-G) this case is similar to the case of (r-L).

(r-G) this case is similar to the case of (l-L).

11.6 Completeness

As before, we need some preliminary lemmas before actually dealing with completeness of the calculus itself.

Lemma 8 ***Weakening***

$$\vdash_{S} \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \;\Rightarrow \vdash_{S} \Pi_{\mathsf{K}} \cup \Pi'_{\mathsf{K}};\Pi_{\mathsf{G}} \cup \Pi'_{\mathsf{G}} \mid \Gamma \cup \Gamma' \supset \Delta \cup \Delta' \mid \Sigma_{\mathsf{L}} \cup \Sigma'_{\mathsf{L}};\Sigma_{\mathsf{F}} \cup \Sigma'_{\mathsf{F}}$$

Proof
We make a proof by induction on the length n of the original proof.

$n = 0$ we have one of the axioms (id), (id-f), or (id-t)

In that case, $\Pi_{\mathsf{K}} \cup \Pi'_{\mathsf{K}};\Pi_{\mathsf{G}} \cup \Pi'_{\mathsf{G}} \mid \Gamma \cup \Gamma' \supset \Delta \cup \Delta' \mid \Sigma_{\mathsf{L}} \cup \Sigma'_{\mathsf{L}};\Sigma_{\mathsf{F}} \cup \Sigma'_{\mathsf{F}}$ is an axiom as well.

$n > 0$ if we still only had an axiom, we have the same situation as before

Otherwise we have to examine the last rule of the proof. If the last rule of the proof was a classical or L/F-rule, the statement follows directly from the premiss and the induction hypothesis.

If the last rule of the proof was a K/G-rule, we get the weakening in Π_{G} and Σ_{F} from the induction hypothesis and the rest can be added to the conclusion of the last step as desired.

This lemma simply states that adding formulas to any of the multisets does not prevent a formula from being provable.

Lemma 9 F ***Invertibility***

$$\vdash_S \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta, \mathsf{F}A \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \quad \Rightarrow \quad \vdash_S \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta, A \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \cup A$$

Proof

We make an induction on the length n of the proof of $\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta, \mathsf{F}A \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$:

$n = 0$ then $\Gamma \supset \Delta, \mathsf{F}A$ is an axiom

Then $\Gamma \supset \Delta$ is an axiom itself, because only propositional variables are allowed in axioms. Then $\Gamma \supset \Delta, A$ is an axiom as well, thus the assumption holds.

$n > 0$ in the case of an axiom, we proceed as before.

Otherwise, one of the rules of the calculus was used in the last step of the proof.

If $\mathsf{F}A$ was not the main formula of the last step of the proof, then we can use the induction hypothesis on the premiss of the rule and then carry through the same step to get to the desired result. In the case of the K/G-rules, we must also used the rule's built-in weakening.

If $\mathsf{F}A$ was the main formula of the last step of the proof, then the assumption is exactly the premiss of the last rule. Thus, the assumption clearly holds.

Lemma 10 ***Strong Invertibility***

The classical and the L/F-rules are strongly invertible.

Proof

(r-$\neg$) $\vdash^n_S \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \neg A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \quad \Rightarrow \quad \vdash^n_S \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$

We make an induction on the length n of the proof:

$n = 0$ in that case, $\Gamma \supset \neg A, \Delta$ is an axiom

Then $\Gamma \supset \Delta$ is an axiom by itself, thus $A, \Gamma \supset \Delta$ is one as well.

$n > 0$ In the case of an axiom, we do the same as before.

Otherwise, a rule was applied. If $\neg A$ was not the main formula of the last step of the proof, we can use the induction hypothesis on the premiss and afterwards carry through the same rule again. This leads to the desired result. In the case of the K/G-rules, we only use the rule's built-in weakening, because $\neg A$ cannot appear in the premiss of one of these rules, it is not the main formula.

If $\neg A$ was the main formula of the last step, then the assumption is equal to the premiss of the rule, and thus must be provable.

(l-$\neg$) This proof is similar to the one for (r-$\neg$).

(r-$\vee$) $\vdash^{n}_{S} \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A \vee B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \quad \Rightarrow \quad \vdash^{n}_{S} \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A, B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$

We make an induction on the length n of the proof:

$n = 0$ in that case, $\Gamma \supset A \vee B, \Delta$ is an axiom

Because $A \vee B$ is not a propositional variable, $\Delta \supset \Gamma$ has to be an axiom by itself. Thus $\Delta \supset A, B, \Gamma$ is an axiom as well.

$n > 0$ In the case of an axiom, we do the same as before.

If $A \vee B$ was not the main formula of the last step of the proof, then we can use the induction hypothesis on the premiss and afterwards carry through the same rule again. This leads to the desired result. In the case of K/G-rules, we only use the rule's built-in weakening, because $A \vee B$ cannot appear in the premiss of one of these rules.

If $A \vee B$ was the main formula of the last step, then the assumption is equal to the premiss of the rule, and thus must be provable.

(l-$\vee$) $\vdash^{n}_{S} \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A \vee B, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \quad \Rightarrow \quad \vdash_{S} \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$ and $\vdash^{n}_{S} \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$

We make an induction on the length n of the proof:

$n = 0$ in that case, $A \vee B, \Delta \supset \Gamma$ is an axiom

Because $A \vee B$ is not a propositional variable, $\Gamma \supset \Delta$ has to be an axiom by itself. Thus $A, \Gamma \supset \Delta$ and $B, \Gamma \supset \Delta$ are axioms as well and the assumption holds.

$n > 0$ In the case of an axiom, we do the same as before.

If $A \vee B$ was not the main formula of the last step of the proof, then we can use the induction hypothesis on the premiss and afterwards carry through the same rule again. This leads to the desired result. In the case of the K/G-rules, we only use the rule's built-in weakening, because $A \vee B$ cannot appear in the premiss of one of these rules.

If $A \vee B$ was the main formula of the last step, then the assumptions are equal to the premisses of the rule, and thus must be provable.

(r-$\wedge$) this proof is similar to the proof of (l-$\vee$).

(l-$\wedge$) this proof is similar to the proof of (r-$\vee$).

(l-$\longrightarrow$) this proof is similar to the proof of (l-$\vee$).

(r-$\longrightarrow$) this proof is similar to the proof of (r-$\vee$).

(l-$\leftrightarrow$) this proof is similar to the proof of (l-$\vee$).

(r-$\leftrightarrow$) this proof is similar to the proof of (l-$\vee$).

(r-L) $\vdash^{n}_{S} \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \mathsf{L}A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \quad \Rightarrow \quad \vdash^{n}_{S} \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A, \Delta \mid \Sigma_{\mathsf{L}} \cup A; \Sigma_{\mathsf{F}}$

We make an induction on the length n of the proof:

$n = 0$ in that case, $\Gamma \supset \mathsf{L}A, \Delta$ is an axiom

Because $\mathsf{L}A$ is not a propositional variable, $\Gamma \supset \Delta$ has to be an axiom by itself. Thus $\Gamma \supset A, \Delta$ is an axiom as well and the assumption holds.

$n > 0$ In the case of an axiom, we do the same as before.

If $\mathsf{L}A$ was not the main formula of the last step of the proof, then we can use the induction hypothesis on the premiss and afterwards carry through the same rule again. This leads to the desired result. In the case of the K/G-rules, we only use the rule's built-in weakening, because $\mathsf{L}A$ cannot appear in the premiss of one of these rules (only the L-formulas stored in Σ_{L} and Π_{K} are considered for these rules).

If $\mathsf{L}A$ was the main formula of the last step, then the assumptions are equal to the premisses of the rule, and thus must be provable.

(r-F) this proof is similar to the proof of (r-L).

(l-K) this proof is similar to the proof of (r-L).

(l-G) this proof is similar to the proof of (r-L).

This lemma shows the improvements of the sequent calculus over the Tait calculus of the previous chapter. While in the Tait calculus only the classical rules were invertible, the L/F-rules are also invertible in the sequent calculus. This greatly enhances the efficiency for automated backward proof search.

Lemma 11 ***Contraction***

a) $\vdash^n_S \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A, A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \quad \Rightarrow \quad \vdash^n_S \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$

b) $\vdash^n_S \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A, A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \quad \Rightarrow \quad \vdash^n_S \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$

Proof

We make a simultaneous induction on the length n of the proofs of $\Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \mid \Gamma \supset A, A, \Delta \mid \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}}$ and $\Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \mid A, A, \Gamma \supset \Delta \mid \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}}$.

$n = 0$ in that case, $\Gamma \supset A, A, \Delta$ and $A, A, \Gamma \supset \Delta$ are axioms.

But then, $\Gamma \supset A, \Delta$ and $A, \Gamma \supset \Delta$ must be axioms as well.

$n > 0$ In the case of an axiom, we do the same as before.

If the main formula of the last step was not the formula A, then we can use the induction hypothesis on the premiss of the rule of the last step and carry through the rule again. This results in the desired assumption.

If the main formula of the last step was A, then we have to make a distinction on the last rule used in the proof. If the rule is invertible (see lemma 10), then we can use the induction hypothesis on the premiss(es), if necessary several times, and then carry through the same rule again.[3]

If the last rule of the proof is not invertible, it has to be one of the K/G-rules. In all these cases, the additional A is dropped in the premiss anyway. Thus, from the same premiss it is possible to get a proof of the assumption.

The lemma states that if some formula appears twice in the multiset, we can remove one appearance without changing provability of the whole sequent.

Lemma 12 ***Axiom Extension***

$$\vdash_S \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A, \Gamma \supset A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \quad \textit{for all formulas } A$$

Proof
We make an induction on the length n of A:

$n = 0$ $A \equiv p_i$, i.e. A is a propositional variable

In that case, the assumption itself is an axiom of the sequent calculus, and thus surely provable.

$n > 0$ we make a distinction between the structure of A:

$A \equiv \neg B$ The induction hypothesis gives us $\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B, \Gamma \supset B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$.
Now we use the following proof:

$$\dfrac{\dfrac{\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B, \Gamma \supset B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset B, \neg B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}}\ (\text{r-}\neg)}{\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \neg B, \Gamma \supset \neg B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}}\ (\text{l-}\neg)$$

$A \equiv B \vee C$ The induction hypothesis gives us $\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B, \Gamma \supset B, C, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$ and $\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid C, \Gamma \supset B, C, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$.
Now we use the following proof:

$$\dfrac{\dfrac{\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B, \Gamma \supset B, C, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \quad \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid C, \Gamma \supset B, C, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B \vee C, \Gamma \supset B, C, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}}\ (\text{l-}\vee)}{\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B \vee C, \Gamma \supset B \vee C, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}}\ (\text{r-}\vee)$$

$A \equiv B \wedge C$ The induction hypothesis gives us $\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B, C, \Gamma \supset B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$ and $\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B, C, \Gamma \supset C, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$.
Now we use the proof:

$$\dfrac{\dfrac{\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B, C, \Gamma \supset B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \quad \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B, C, \Gamma \supset C, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B, C, \Gamma \supset B \wedge C, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}}\ (\text{r-}\wedge)}{\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B \wedge C, \Gamma \supset B \wedge C, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}}\ (\text{l-}\wedge)$$

[3] the length of the proof is not changed by using the induction hypothesis or by using the strong invertibility of a rule.

$A \equiv B \to C$ The induction hypothesis gives us $\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B,\Gamma \supset B,C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}$ and $\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B,C,\Gamma \supset C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}$.

Now we use the proof:

$$\dfrac{\dfrac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B,\Gamma \supset B,C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \quad \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B,C,\Gamma \supset C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B,B \to C,\Gamma \supset C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\,(\text{l-}\to)}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B \to C,\Gamma \supset B \to C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\,(\text{r-}\to)$$

$A \equiv B \leftrightarrow C$ The induction hypothesis gives us

$$\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B,\Gamma \supset B,C,C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}},$$

$$\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B,B,C,\Gamma \supset C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}},$$

$$\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid C,\Gamma \supset B,B,C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}},\ \text{and}$$

$$\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid C,B,C,\Gamma \supset B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}.$$

Now we use the proof:

$$[A]\text{:}\ \dfrac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B,\Gamma \supset B,C,C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \quad \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B,B,C,\Gamma \supset C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B,B \leftrightarrow C,\Gamma \supset C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\,(\text{l-}\leftrightarrow)$$

$$\dfrac{[A] \quad \dfrac{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid C,\Gamma \supset B,B,C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \quad \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid C,B,C,\Gamma \supset B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid C,B \leftrightarrow C,\Gamma \supset B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\,(\text{l-}\leftrightarrow)}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid B \leftrightarrow C,\Gamma \supset B \leftrightarrow C,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\,(\text{r-}\leftrightarrow)$$

$A \equiv \mathsf{L}B$ The induction hypothesis gives us $\emptyset;\Pi_{\mathsf{G}} \mid B,\Pi_{\mathsf{K}},\Pi_{\mathsf{G}} \supset B,\Sigma_{\mathsf{L}},\Sigma_{\mathsf{F}} \mid \emptyset;\Sigma_{\mathsf{F}}$.

Now we use the proof:

$$\dfrac{\dfrac{\emptyset;\Pi_{\mathsf{G}} \mid B,\Pi_{\mathsf{K}},\Pi_{\mathsf{G}} \supset B,\Sigma_{\mathsf{L}},\Sigma_{\mathsf{F}} \mid \emptyset;\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{L}B,\Gamma \supset B,\Delta \mid \Sigma_{\mathsf{L}} \cup B;\Sigma_{\mathsf{F}}}\,(\text{l-L})}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{L}B,\Gamma \supset \mathsf{L}B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\,(\text{r-L})$$

$A \equiv \mathsf{F}B$ The induction hypothesis gives us $\emptyset;\Pi_{\mathsf{G}} \mid B,\Pi_{\mathsf{G}} \supset B,\Sigma_{\mathsf{F}} \mid \emptyset;\Sigma_{\mathsf{F}}$.

Now we use the proof:

$$\dfrac{\dfrac{\emptyset;\Pi_{\mathsf{G}} \mid B,\Pi_{\mathsf{G}} \supset B,\Sigma_{\mathsf{F}} \mid \emptyset;\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{F}B,\Gamma \supset B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \cup B}\,(\text{l-F})}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{F}B,\Gamma \supset \mathsf{F}B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\,(\text{r-F})$$

$A \equiv \mathsf{K}B$ The induction hypothesis gives us $\emptyset;\Pi_{\mathsf{G}} \mid B,\Pi_{\mathsf{K}},\Pi_{\mathsf{G}} \supset B,\Sigma_{\mathsf{L}},\Sigma_{\mathsf{F}} \mid \emptyset;\Sigma_{\mathsf{F}}$.

Now we use the proof:

$$\dfrac{\dfrac{\emptyset;\Pi_{\mathsf{G}} \mid B,\Pi_{\mathsf{K}},\Pi_{\mathsf{G}} \supset B,\Sigma_{\mathsf{L}},\Sigma_{\mathsf{F}} \mid \emptyset;\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}} \cup B;\Pi_{\mathsf{G}} \mid B,\Gamma \supset \mathsf{K}B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\,(\text{r-K})}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{K}B,\Gamma \supset \mathsf{K}B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\,(\text{l-K})$$

$A \equiv \mathsf{G}B$ The induction hypothesis gives us $\emptyset;\Pi_{\mathsf{G}} \mid B,\Pi_{\mathsf{G}} \supset B,\Sigma_{\mathsf{F}} \mid \emptyset;\Sigma_{\mathsf{F}}$.

Now we use the proof:

$$\dfrac{\dfrac{\emptyset;\Pi_{\mathsf{G}} \mid B,\Pi_{\mathsf{G}} \supset B,\Sigma_{\mathsf{L}} \mid \emptyset;\Sigma_{\mathsf{F}}}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \cup B \mid B,\Gamma \supset \mathsf{G}B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\,(\text{r-G})}{\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{G}B,\Gamma \supset \mathsf{G}B,\Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}}\,(\text{l-G})$$

This lemma allows us to directly use complex formulas in the place of the propositional variables in axioms. This allows us to end proof search much earlier and thus speeds up proving. Directly using complex formulas in the axioms of the original calculus, on the other hand, would make most of the proofs in this chapter much more complicated.

Now, we are ready to prove the completeness of the sequent calculus. We do that by showing that every proof of the Tait Calculus of the previous chapter can be transformed into a proof of the sequent calculus.

Theorem 8 ***Completeness***

$$\vdash_T \mathrm{nnf}(\Delta) \quad \Rightarrow \quad \vdash_S \emptyset; \emptyset \mid \quad \supset \Delta \mid \emptyset; \emptyset$$

Proof
We make an induction on the length n of the proof in the Tait calculus.

$n = 0$ then Γ is an Axiom, i.e. $\Gamma = A, \neg A, \Gamma'$ for some formula A

We now make the following proof in the sequent calculus:

$$\frac{\emptyset; \emptyset \mid A \supset A, \Gamma' \mid \emptyset; \emptyset}{\emptyset; \emptyset \mid \quad \supset A, \neg A, \Gamma' \mid \emptyset; \emptyset} \text{ (r-}\neg\text{)}$$

Using lemma 12 we get the proof.

$n > 0$ If we still have an axiom, we use the same as before.

Otherwise, we may assume that Γ was deduced by one of the rules of the Tait calculus. We assume that the formula A was the main formula of the last rule used in the proof and distinguish the different cases by the last rule used.

($\vee$) then $A \equiv B \vee C$ and from the last step we have $\vdash_T^{<n} B, C, \Gamma'$
With the induction hypothesis we get $\vdash_S \emptyset; \emptyset \mid \quad \supset A, B, \Gamma \mid \emptyset; \emptyset$. Thus, we can make the following proof in the sequent calculus:

$$\frac{\emptyset; \emptyset \mid \quad \supset A, B, \Gamma' \mid \emptyset; \emptyset}{\emptyset; \emptyset \mid \quad \supset A \vee B, \Gamma' \mid \emptyset; \emptyset} \text{ (r-}\vee\text{)}$$

($\wedge$) then $A \equiv B \wedge C$ and from the last step we have $\vdash_T^{<n} B, \Gamma'$ and $\vdash_T^{<n} C, \Gamma'$
With the induction hypothesis (twice) we get $\vdash_S \emptyset; \emptyset \mid \quad \supset B, \Gamma' \mid \emptyset; \emptyset$ and $\vdash_S \emptyset; \emptyset \mid \quad \supset C, \Gamma' \mid \emptyset; \emptyset$. Thus, we can make the following proof in the sequent calculus:

$$\frac{\emptyset; \emptyset \mid \quad \supset B, \Gamma' \mid \emptyset; \emptyset \qquad \emptyset; \emptyset \mid \quad \supset C, \Gamma' \mid \emptyset; \emptyset}{\emptyset; \emptyset \mid \quad \supset B \wedge C, \Gamma' \mid \emptyset; \emptyset} \text{ (r-}\wedge\text{)}$$

(L) then $A \equiv \mathsf{L}B$ and from the last step we have $\vdash^{\leq n}_{T} A, \Gamma'$

With the induction hypothesis we get $\vdash_S \emptyset; \emptyset \mid \ \supset B, \Gamma' \mid \emptyset; \emptyset$. Using Lemma 8 this can be weakened to $\vdash_S \emptyset; B, \Gamma' \mid \ \supset \ \mid A; \emptyset$. Now, we can make the following proof in the sequent calculus:

$$\dfrac{\emptyset; \emptyset \mid \ \supset B, \Gamma' \mid B; \emptyset}{\emptyset; \emptyset \mid \ \supset \mathsf{L}B, \Gamma' \mid \emptyset; \emptyset}\ (\text{r-}\mathsf{L})$$

(F) then $A \equiv \mathsf{F}B$ and from the last step we have $\vdash^{\leq n}_{T} B, \Gamma'$

With the induction hypothesis we get $\vdash_S \emptyset; \emptyset \mid \ \supset B, \Gamma' \mid \emptyset; \emptyset$. Using Lemma 8 this can be weakened to $\vdash_S \emptyset; B, \Gamma' \mid \ \supset \ \mid \emptyset; B$. Now, we can make the following proof in the sequent calculus:

$$\dfrac{\emptyset; \emptyset \mid \ \supset B, \Gamma' \mid \emptyset; B}{\emptyset; \emptyset \mid \ \supset \mathsf{F}B, \Gamma' \mid \emptyset; \emptyset}\ (\text{r-}\mathsf{F})$$

(K) then $A \equiv \mathsf{K}B$ and from the last step we have $\vdash^{\leq n}_{T} B, \Gamma'_{\mathsf{L}}, \Gamma'_{\mathsf{F}}, \mathsf{F}\Gamma'_{\mathsf{F}}$.

With the induction hypothesis we get $\vdash_S \emptyset; \emptyset \mid \ \supset B, \Gamma'_{\mathsf{L}}, \Gamma'_{\mathsf{F}}, \mathsf{F}\Gamma'_{\mathsf{F}} \mid \emptyset; \emptyset$. Using lemma 9 once for each member of Γ'_{F}, we get $\vdash_S \emptyset; \emptyset \mid \ \supset B, \Gamma'_{\mathsf{L}}, \Gamma'_{\mathsf{F}}, \Gamma'_{\mathsf{F}} \mid \emptyset; \Delta$. Now we can use contraction (lemma 11) to achieve $\vdash_S \emptyset; \emptyset \mid \ \supset B, \Gamma'_{\mathsf{L}}, \Gamma'_{\mathsf{F}} \mid \emptyset; \Gamma'_{\mathsf{F}}$. We can now make the following proof in the sequent calculus:

$$\begin{array}{cl}
\emptyset; \emptyset \mid \ \supset B, \Gamma'_{\mathsf{L}}, \Gamma'_{\mathsf{F}} \mid \emptyset; \Gamma'_{\mathsf{F}} & \\
\hline
\emptyset; \emptyset \mid \ \supset \mathsf{K}B, \Gamma', \Gamma'_{\mathsf{L}}, \Gamma'_{\mathsf{F}} \mid \Gamma'_{\mathsf{L}}; \Gamma'_{\mathsf{F}} & (\text{r-}\mathsf{K}) \\
\hline
\emptyset; \emptyset \mid \ \supset \mathsf{K}B, \mathsf{F}\Gamma'_{\mathsf{F}}, \Gamma', \Gamma'_{\mathsf{L}} \mid \Gamma'_{\mathsf{L}}; \emptyset & (\text{r-}\mathsf{F}\text{ several times}) \\
\hline
\emptyset; \emptyset \mid \ \supset \mathsf{K}B, \mathsf{L}\Gamma'_{\mathsf{L}}, \mathsf{F}\Gamma'_{\mathsf{F}}, \Gamma' \mid \emptyset; \emptyset & (\text{r-}\mathsf{L}\text{ several times})
\end{array}$$

(G) then $A \equiv \mathsf{G}B$ and from the last step we have $\vdash^{\leq n}_{T} B, \mathsf{F}\Gamma'_{\mathsf{F}}$.

With the induction hypothesis we get $\vdash_S \emptyset; \emptyset \mid \ \supset B, \mathsf{F}\Gamma'_{\mathsf{F}} \mid \emptyset; \emptyset$. Using lemma 9 once for each member of $\mathsf{F}\Gamma'_{\mathsf{F}}$, we get $\vdash_S \emptyset; \emptyset \mid \ \supset B, \Gamma'_{\mathsf{F}} \mid \emptyset; \Gamma'_{\mathsf{F}}$. We can now make the following proof in the sequent calculus:

$$\begin{array}{cl}
\emptyset; \emptyset \mid \ \supset B, \Gamma'_{\mathsf{F}} \mid \emptyset; \Gamma'_{\mathsf{F}} & \\
\hline
\emptyset; \emptyset \mid \ \supset \mathsf{G}B, \Gamma'_{\mathsf{F}}, \Gamma' \mid \emptyset; \Gamma'_{\mathsf{F}} & (\text{r-}\mathsf{G}) \\
\hline
\emptyset; \emptyset \mid \ \supset \mathsf{G}B, \mathsf{F}\Gamma'_{\mathsf{F}}, \Gamma' \mid \emptyset; \emptyset & (\text{r-}\mathsf{F}\text{ several times})
\end{array}$$

11.7 Properties

As before with the Tait calculus, we take a look at some properties of the sequent calculus.

11.7.1 Subformula Property

It can be easily seen that all rules of the sequent calculus have the subformula property, i.e. all formulas present in the premiss of a rule are subformulas of one of the formulas of the conclusion.

11.7.2 Contraction

Contraction has been proven in lemma 11 and is thus valid in the sequent calculus as well.

Invertible Rules

As shown in lemma 10, most rules in the sequent calculus are strongly invertible. The rest of the rules, namely the K/G-rules are not invertible, as was the case in the Tait calculus.

11.7.3 Backward Proof Search

A formula A is provable in the sequent calculus, written as

$$\vdash_S A$$

if and only if there exists a derivation sequence which starts at axioms, ends with the formula A and only uses the rules of the calculus. This is exactly the same as for the Tait calculus. Again, not every derivation ends in axioms only, even if the formula is provable. Consider the following example of the proof of the formula $\mathsf{F}\neg B \vee \mathsf{G}A \vee \mathsf{G}B$ (which is the same as $\neg\mathsf{G}B \vee \mathsf{G}A \vee \mathsf{G}B$, thus provable):

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\emptyset;\emptyset \mid B \supset A \mid \emptyset;\neg B}{\emptyset;\emptyset \mid \ \supset \neg B, A \mid \emptyset;\neg B}\ (\text{r-}\neg)}{\emptyset;\emptyset \mid \ \supset \neg B, \mathsf{G}A, \mathsf{G}B \mid \emptyset;\neg B}\ (\text{r-G})}{\emptyset;\emptyset \mid \ \supset \mathsf{F}\neg B, \mathsf{G}A, \mathsf{G}B \mid \emptyset;\emptyset}\ (\text{r-F})}{\emptyset;\emptyset \mid \ \supset \mathsf{F}\neg B, \mathsf{G}A \vee \mathsf{G}B \mid \emptyset;\emptyset}\ (\text{r-}\vee)}{\emptyset;\emptyset \mid \ \supset \mathsf{F}\neg B \vee \mathsf{G}A \vee \mathsf{G}B \mid \emptyset;\emptyset}\ (\text{r-}\vee)$$

This proof does not work, i.e. we come to a position where we can't go on. The problem comes from the wrong selection of the main formula for the (G)-rule. If we use $\mathsf{G}B$ instead of $\mathsf{G}A$ as main formula, we get a successful proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\emptyset;\emptyset \mid B \supset B \mid \emptyset;\neg B}{\emptyset;\emptyset \mid \ \supset \neg B, B \mid \emptyset;\neg B}\,(\text{r-}\neg)}{\emptyset;\emptyset \mid \ \supset \neg B, \mathsf{G}A, \mathsf{G}B \mid \emptyset;\neg B}\,(\text{r-}\mathsf{G})}{\emptyset;\emptyset \mid \ \supset \mathsf{F}\neg B, \mathsf{G}A, \mathsf{G}B \mid \emptyset;\emptyset}\,(\text{r-}\mathsf{F})}{\emptyset;\emptyset \mid \ \supset \mathsf{F}\neg B, \mathsf{G}A \vee \mathsf{G}B \mid \emptyset;\emptyset}\,(\text{r-}\vee)}{\emptyset;\emptyset \mid \ \supset \mathsf{F}\neg B \vee \mathsf{G}A \vee \mathsf{G}B \mid \emptyset;\emptyset}\,(\text{r-}\vee)$$

Thus, the situation is similar to the situation of the Tait calculus. This time, we have more invertible rule. Because these rules don't need backtracking, this calculus prospects to be more efficient that the Tait calculus. Furthermore, because of the double-sided nature of the calculus, the classical implication and equivalence can be treated much more efficiently. Instead of having to replace them with their definition, we can directly treat them.

11.7.4 Termination

The given sequent calculus does not have the same problems concerning termination as the Tait calculus presented in the previous chapter. The (L)- and (F)-rules are specifically made to treat an L- or F-formula only once.[4] Thus a loop cannot appear in such a case.

Unfortunately, there is a drawback to this solution. Each time a G-formula is treated, all previously handled F-formulas are brought back for treatment. If there is a G-formula present in the sequent, then we can create a loop. Consider the following derivation of the formula F GA:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\emptyset;\emptyset \mid \ \supset A, \mathsf{G}A \mid \emptyset;\mathsf{G}A}\,(\text{r-}\mathsf{G})}{\emptyset;\emptyset \mid \ \supset A, \mathsf{G}A \mid \emptyset;\mathsf{G}A}\,(\text{r-}\mathsf{G})}{\emptyset;\emptyset \mid \ \supset A, \mathsf{G}A \mid \emptyset;\mathsf{G}A}\,(\text{r-}\mathsf{G})}{\emptyset;\emptyset \mid \ \supset \mathsf{FG}A \mid \emptyset;\emptyset}\,(\text{r-}\mathsf{F})$$

Thus, without taking special care, this calculus is bound to loop just as the Tait calculus. This problem will be tackled in the next section.

[4] that is actually once in each state treated; the application of a (K)- or (G)-rule makes it possible that such a formula is used again.

11.7.5 Disjunction Principle

Lemma 13 ***Disjunction Principle***

$$
\begin{array}{l}
\vdash_S \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{F}A_1,\ldots,\mathsf{F}A_n,\mathsf{L}B_1,\ldots,\mathsf{L}B_m \supset \mathsf{G}C_1,\ldots,\mathsf{G}C_k,\mathsf{K}D_1,\ldots,\mathsf{K}D_l \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \\
\quad \Rightarrow \;\; \vdash_S \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{F}A_1 \supset \quad \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \;\; \textit{or} \;\; \ldots \;\; \textit{or} \;\; \vdash_S \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{F}A_n \supset \quad \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \;\; \textit{or} \\
\qquad\quad \vdash_S \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{L}B_1 \supset \quad \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \;\; \textit{or} \;\; \ldots \;\; \textit{or} \;\; \vdash_S \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{K}B_m \supset \quad \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \;\; \textit{or} \\
\qquad\quad \vdash_S \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \quad \supset \mathsf{G}C_1 \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \;\; \textit{or} \;\; \ldots \;\; \textit{or} \;\; \vdash_S \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \quad \supset \mathsf{G}C_k \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \;\; \textit{or} \\
\qquad\quad \vdash_S \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \quad \supset \mathsf{K}D_1 \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \;\; \textit{or} \;\; \ldots \;\; \textit{or} \;\; \vdash_S \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \quad \supset \mathsf{K}D_l \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}
\end{array}
$$

Proof
Let $\vdash_S \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{F}A_1,\ldots,\mathsf{F}A_n,\mathsf{L}B_1,\ldots,\mathsf{L}B_m \supset \mathsf{G}C_1,\ldots,\mathsf{G}C_k,\mathsf{K}D_1,\ldots,\mathsf{K}D_l \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}$

This can't be an axiom, because we only allow propositional variables in axioms. Thus a rule with main formula A was applied last.

We only treat the case where $A \equiv \mathsf{F}A_i$. All other cases can be done similarly.

The premiss of the rule gives us $\vdash_S \emptyset;\Pi_{\mathsf{G}} \mid A_i,\Pi_{\mathsf{K}},\Pi,\mathsf{G} \supset \Sigma_{\mathsf{L}},\Sigma_{\mathsf{K}} \mid \emptyset;\Sigma_{\mathsf{F}}$. Using the (l-F)-rule without built-in weakening gives us $\vdash_S \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \mathsf{F}A_i \supset \quad \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}$. This exactly what we want to prove.

🐧

Again, this property is crucial for proof search. As with the Tait Calculus, it allows to treat backtracking formulas independent from each other.

11.8 Loop-Check

As mentioned, proof search in this sequent calculus does not always terminate. In order to be able to implement the calculus in a theorem prover, termination has to be guaranteed. We guarantee termination by adding a loop-check to the calculus.

11.8.1 Preparations

Before we actually deal with the loop check we have to make some preparations.

Lemma 14 ***Side Formulas***
If $\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}$ *is the conclusion of a classical or* $\mathsf{L}/\!\mathsf{F}$ *rule and* $\Pi'_{\mathsf{L}};\Pi'_{\mathsf{F}} \mid \Gamma' \supset \Delta' \mid \Sigma'_{\mathsf{L}};\Sigma'_{\mathsf{F}}$ *one of its premisses and* A *is not the main formula of the rule, then*

$$A \in \Delta \quad \Rightarrow \quad A \in \Delta'$$

and

$$A \in \Gamma \quad \Rightarrow \quad A \in \Gamma'$$

Proof

A closer look at the rules in question shows, that all these rules take over all side formulas from the conclusion to the premiss.

This lemma just says that side formulas are retained with classical and L/F-rules.

Lemma 15 ***Marked Side Formulas***

If $\Pi_\mathsf{K}; \Pi_\mathsf{G} \mid \Gamma \supset \Delta \mid \Sigma_\mathsf{L}; \Sigma_\mathsf{F}$ *is the conclusion of a rule and* $\Pi'_\mathsf{L}; \Pi'_\mathsf{F} \mid \Gamma' \supset \Delta' \mid \Sigma'_\mathsf{L}; \Sigma'_\mathsf{F}$ *one of its premisses and* A *is not the main formula of the rule, then we have*

$$A \in \Sigma_\mathsf{F} \text{ and } A \in \Delta \quad \Rightarrow \quad A \in \Delta'$$

and

$$A \in \Pi_\mathsf{G} \text{ and } A \in \Gamma \quad \Rightarrow \quad A \in \Gamma'$$

Proof

We have to take a closer look at the rules of the calculus.

We don't have to check the axiom rules, because they don't have any premisses. All the classical rules take over all side formulas that are present in the conclusion (lemma 14). Thus our proposition surly holds for these rules. The same is true for the L/F rules.

What is left are the K/G rules. For these rules we use the fact, that $A \in \Sigma_\mathsf{F}$. For all these rules, Σ_F and thus A is present in the premiss. Thus, our proposition holds for these rules as well.

For the second statement we can argue similarly.

Definition 28 ***Logical Element***

Γ *and* Δ *are arbitrary, finite multisets. Then we define* $[\Gamma;\Delta]^n$ *inductively by*

$$\begin{array}{lclcllll}
[\Gamma;\Delta]^0 & = & \Delta, & & & & & \\
[\Gamma;\Delta]^{n+1} & = & [\Gamma;\Delta]^n & \cup & \{\neg B & \mid & B \in [\Delta;\Gamma]^n\} & \\
 & & & \cup & \{B \vee C & \mid & B \in [\Gamma;\Delta]^n & \text{and} \quad C \in [\Gamma;\Delta]^n\} \\
 & & & \cup & \{B \wedge C & \mid & B \in [\Gamma;\Delta]^n & \text{or} \quad C \in [\Gamma;\Delta]^n\} \\
 & & & \cup & \{B \rightarrow C & \mid & B \in [\Delta;\Gamma]^n & \text{and} \quad C \in [\Gamma;\Delta]^n\} \\
 & & & \cup & \{B \leftrightarrow C & \mid & (B \in [\Delta;\Gamma]^n & \text{and} \quad C \in [\Gamma;\Delta]^n) \text{ or} \\
 & & & & & & (B \in [\Gamma;\Delta]^n & \text{and} \quad C \in [\Delta;\Gamma]^n)\} \\
 & & & \cup & \{\mathsf{L}B & \mid & B \in [\Gamma;\Delta]^n\} & \\
 & & & \cup & \{\mathsf{F}B & \mid & B \in [\Gamma;\Delta]^n\} &
\end{array}$$

We say a formula A is a logical element *of* $\Gamma; \Delta$*, written as* $A \dot{\in} \Gamma; \Delta$ *iff there is an* n *with* $A \in [\Gamma; \Delta]^n$.

It has to be noted, that if a G-formula is in the set, then it's subformulas are not necessarily as well. Thus $\mathsf{G}A \dot{\in} \Gamma; \Delta$ does not require that $A \dot{\in} \Gamma; \Delta$.

This definition allows to say that a formula is present in a sequent, maybe after some proof steps were carried through and the formula was broken up. The multiset Γ represents the left side of the sequent and Δ represents the right side.

Lemma 16 ***Properties of*** $\dot{\in}$

1. $A \dot{\in} \Gamma; \Delta \quad \Rightarrow \quad \neg A \dot{\in} \Delta; \Gamma$
2. $A \dot{\in} \Gamma; \Delta$ *and* $B \dot{\in} \Gamma; \Delta \quad \Rightarrow \quad A \vee B \dot{\in} \Gamma; \Delta$
3. $A \dot{\in} \Gamma; \Delta$ *or* $B \dot{\in} \Gamma; \Delta \quad \Rightarrow \quad A \wedge B \dot{\in} \Gamma; \Delta$
4. $A \dot{\in} \Delta; \Gamma$ *and* $B \dot{\in} \Gamma; \Delta \quad \Rightarrow \quad A \rightarrow B \dot{\in} \Gamma; \Delta$
5. $(A \dot{\in} \Delta; \Gamma$ *and* $B \dot{\in} \Gamma; \Delta)$ *or* $(B \dot{\in} \Delta; \Gamma$ *and* $A \dot{\in} \Gamma; \Delta) \quad \Rightarrow \quad A \leftrightarrow B \dot{\in} \Gamma; \Delta$
6. $A \dot{\in} \Gamma; \Delta \quad \Rightarrow \quad \mathsf{L}A \dot{\in} \Gamma; \Delta$
7. $A \dot{\in} \Gamma; \Delta \quad \Rightarrow \quad \mathsf{F}A \dot{\in} \Gamma; \Delta$

Proof
These propositions directly follow from the definition above.

The properties of $\dot{\in}$ will be used in the proofs below.

Lemma 17 ***Weakening for*** $\dot{\in}$
If $\Delta \subset \Delta'$ *and* $\Gamma \subset \Gamma'$ *then*

$$A \dot{\in} \Gamma; \Delta \quad \Rightarrow \quad A \dot{\in} \Gamma'; \Delta'$$

Proof
This directly follows from the definition of $\dot\in$, or more precise the definition of $[\Gamma;\Delta]^n$. We surely have $[\Gamma;\Delta]^0 \subset [\Gamma';\Delta']^0$. The definition also gives us $[\Gamma;\Delta]^n \subset [\Gamma';\Delta']^n$ for each n.

From $A \mathrel{\dot\in} \Gamma;\Delta$ we know that there is an n with $A \in [\Gamma;\Delta]^n$. But then also $A \in [\Gamma';\Delta']^n$, i.e. $A \mathrel{\dot\in} \Gamma';\Delta'$.

With this lemma, we are allowed to enlarge the multisets without changing the fact that a formula is a logical element.

The following has to be noted:

$$B \in \text{subfml}(A) \quad \not\Rightarrow \quad B \mathrel{\dot\in} \emptyset; A$$

This means that not all subformulas of a formula are logical elements of its multisets.

Consider the case that $A \equiv B \wedge C$. In the case that $C \mathrel{\dot\in} \emptyset; A$ there is no guarantee that $B \mathrel{\dot\in} \emptyset; A$ as well. The definition only requires that either B or C must be, but not both.

Lemma 18 ***Main Formulas as Logical Elements***
If $\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}$ is the conclusion of a classical or $\mathsf{L/F}$ rule and $\Pi'_{\mathsf{L}};\Pi'_{\mathsf{F}} \mid \Gamma' \supset \Delta' \mid \Sigma'_{\mathsf{L}};\Sigma'_{\mathsf{F}}$ one of its premisses and A is the main formula of this rule, then we have

$$A \in \Delta \quad \Rightarrow \quad A \mathrel{\dot\in} \Gamma';\Delta'$$

and

$$A \in \Gamma \quad \Rightarrow \quad A \mathrel{\dot\in} \Delta';\Gamma'$$

Proof
We only prove the first part. The second part can be proven similarly.

We distinguish several cases for the different rules that could be used:

$A \equiv \neg B$	In that case, $B \in \Gamma'$ and thus $B \mathrel{\dot\in} \Delta';\Gamma'$. With lemma 16 (1), we get $A \equiv \neg B \in \Gamma';\Delta'$.
$A \equiv B \vee C$	The rule gives us that $B \in \Delta'$ and $C \in \Delta$. Thus, from the definition of $\dot\in$ we have $B \mathrel{\dot\in} \Gamma';\Delta'$ and $C \mathrel{\dot\in} \Gamma';\Delta'$. With lemma 16 (2), this gives us the proposition.
$A \equiv B \wedge C$	Depending on which premiss we choose, we either have $B \in \Delta'$ or $C \in \Delta'$. That means, either $B \mathrel{\dot\in} \Gamma';\Delta'$ or $C \mathrel{\dot\in} \Gamma';\Delta'$. According to lemma 16 (3), this gives us the proposition.
$\ldots$	The proofs for the remaining cases can be done similarly to the ones above and thus are not detailed here.

This lemma says that main formulas of classical and L/F-rules are logical elements of the premiss of the rule. This is extended in the next lemma to all formulas.

Lemma 19 ***Formulas as Logical Elements***
If $\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$ *is the conclusion of a classical or* L/F *rule and* $\Pi'_{\mathsf{L}}; \Pi'_{\mathsf{F}} \mid \Gamma' \supset \Delta' \mid \Sigma'_{\mathsf{L}}; \Sigma'_{\mathsf{F}}$ *one of its premisses then*

$$A \mathrel{\dot{\in}} \Gamma; \Delta \quad \Rightarrow \quad A \mathrel{\dot{\in}} \Gamma'; \Delta'$$

and

$$A \mathrel{\dot{\in}} \Delta; \Gamma \quad \Rightarrow \quad A \mathrel{\dot{\in}} \Delta'; \Gamma'$$

Proof
If A is the main formula of the proof, then, according to lemma 18, we are already done.

Otherwise, lemma 15 directly gives us the desired proposition.

With this lemma, every formula of a conclusion is a logical element of the premiss. The definition of $\dot{\in}$ was actually tailored to make this lemma possible.

Lemma 20 ***Limited Marked Formulas as Logical Elements***
If $\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$ *is the conclusion of a rule and* $\Pi'_{\mathsf{L}}; \Pi'_{\mathsf{F}} \mid \Gamma' \supset \Delta' \mid \Sigma'_{\mathsf{L}}; \Sigma'_{\mathsf{F}}$ *one of its premisses, then we have the following*

$$A \in \Sigma_{\mathsf{F}} \text{ and } A \mathrel{\dot{\in}} \Gamma; \Delta \quad \Rightarrow \quad A \mathrel{\dot{\in}} \Gamma'; \Delta'$$

and

$$A \in \Pi_{\mathsf{G}} \text{ and } A \mathrel{\dot{\in}} \Delta; \Gamma \quad \Rightarrow \quad A \mathrel{\dot{\in}} \Delta'; \Gamma'$$

Proof
We assume $A \in \Sigma_{\mathsf{F}}$ and $A \mathrel{\dot{\in}} \Gamma; \Delta$. If A is the main formula of the rule, then according to lemma 18 we directly get the proposition. Thus, we only have to treat the cases where A is not the main formula of the rule.

If the rule in question was either a classical or a L/F-rule, then lemma 19 directly gives us the proposition, without using that $A \in \Sigma_{\mathsf{F}}$ or $A \in \Pi_{\mathsf{G}}$.

What's left, is to show the proposition for the K/G rules. But for these rules we have—because of $A \in \Sigma_{\mathsf{F}}$—that $A \in \Delta'$ and thus the proposition surely holds.

Lemma 21 ***Marked Formulas as Logical Elements***

For each conclusion $\Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}$ *and premiss* $\Pi'_{\mathsf{L}};\Pi'_{\mathsf{F}} \mid \Gamma' \supset \Delta' \mid \Sigma'_{\mathsf{L}};\Sigma'_{\mathsf{F}}$ *of a rule in a proof of the sequent* $\emptyset;\emptyset \mid \ \supset B \mid \emptyset;\emptyset$*, we have*

$$A \in \Sigma_{\mathsf{F}} \quad \Rightarrow \quad A \mathrel{\dot{\in}} \Gamma';\Delta'$$

and

$$A \in \Pi_{\mathsf{G}} \quad \Rightarrow \quad A \mathrel{\dot{\in}} \Delta';\Gamma'$$

Proof

We only prove the first statement, because the proof of the second is similar.

If the rule in question is a K/G rule, then we surely have from $A \in \Sigma_{\mathsf{F}}$ that $A \in \Delta'$, thus the conclusion holds.

If $A \mathrel{\dot{\in}} \Gamma;\Delta$ then we know from lemma 20 that also $A \mathrel{\dot{\in}} \Gamma';\Delta'$.

What's left is the case that $A \mathrel{\dot{\notin}} \Gamma;\Delta$. If $A \mathrel{\dot{\in}} \Gamma'',\Delta''$ of a $\Gamma'';\Delta''$ of an earlier rule, then we'd have $A \mathrel{\dot{\in}} \Gamma;\Delta$ as well (lemma 20). Thus, A was never a logical element of a rule. But then, $\mathsf{F}A$ never was the main formula of a rule, because otherwise A must be an element of its premiss. But then we cannot have $A \in \Sigma_{\mathsf{F}}$, because we started with empty sets for the marked formulas. Thus, this case never occurs.

Lemma 22 ***Removing Formulas***

$$A \in \Sigma_{\mathsf{F}} \text{ and } A \mathrel{\dot{\in}} \Gamma;\Delta \text{ and } \vdash_{S} \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \Delta, A \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \quad \Rightarrow \quad \vdash_{S} \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}$$

and

$$A \in \Pi_{\mathsf{G}} \text{ and } A \mathrel{\dot{\in}} \Delta;\Gamma \text{ and } \vdash_{S} \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma, A \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}} \quad \Rightarrow \quad \vdash_{S} \Pi_{\mathsf{K}};\Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}};\Sigma_{\mathsf{F}}$$

Proof
We make a proof by induction over the structure of the proof:

If $\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta, A \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$ is an axiom, then we can have one of two cases. In the first case, $\Gamma \supset \Delta$ is an axiom by itself. Then the proposition surely holds. Otherwise, the sequent is an axiom because $\neg A \in \Gamma$. But then A must be a propositional variable. Thus, from $A \dot{\in} \Delta; \Gamma$ follows that $A \in \Delta$. But then the proposition surely is an axiom as well.

If A is not the main formula of the last step of the proof, then in the case of classical or L/F rules, A is present in the premiss as well and we can use the induction hypothesis. In the case of a K/G rule A is not present in the premiss and we can directly prove our proposition using the built-in weakening.

What's left is the case that A is the main formula of the last step of the proof. We distinguish different cases by the structure of A.

$A \equiv \neg B$	The premiss of the proof gives us $\vdash_{S} \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma, B \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$. But either we had from $A \dot{\in} \Gamma; \Delta$ that $A \in \Delta$, which would directly allow us to get the proposition, or we have $B \dot{\in} \Delta; \Gamma$. But then we can use the induction hypothesis on the second part to get the proposition.
$\ldots$	For the other classical and L/F rules we can give similar proofs.
$A \equiv \mathsf{G}B$	Because we know that $A \dot{\in} \Gamma; \Delta$, we know from the definition of '$\dot{\in}$' that $A \in \Delta$. But then the proposition directly follows using contraction.
$A \equiv \mathsf{K}B$	This case is similar to the previous one.

The proof of the second part of the lemma is done similarly.

Theorem 9 ***Removing*** $\mathsf{F}A$
If $A \in \Sigma_{\mathsf{F}}$ *then*

$$\vdash_{S} \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta, \mathsf{F}A \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \quad \Rightarrow \quad \vdash_{S} \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$$

Proof
We make an induction on the structure of the proof in question:

If we have an axiom, it can surely not be because of $\mathsf{F}A$, because we only allow propositional variables in axioms. Thus we can remove $\mathsf{F}A$ and still have an axiom.

If $\mathsf{F}A$ was not the main formula of the rule, then we can use the induction hypothesis on the premiss to get the desired result.

We are left with the case that $\mathsf{F}A$ is the main formula of the rule. But then the premiss of the rule is $\Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta, A \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$. With lemma 22[5] follows $\vdash_{S} \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$. This is exactly what we want to prove.

Corollary 2 ***Removing*** $\mathsf{G}A$
If $A \in \Pi_{\mathsf{G}}$ *then*

$$\vdash_S \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \mathsf{G}A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \quad \Rightarrow \quad \vdash_S \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}}$$

Proof
The proof of this lemma can be done analogously as the one for the theorem.

The theorem shows us that it is completely enough to have an F rule that treats new F formulas and that it's not necessary to treat F formulas again if they were already treated once. Using theorem 9 allows us to add the condition '$A \notin \Gamma_{\mathsf{F}}$' to the ($\mathsf{F}$)-rules, without having to add a rule for $A \in \Gamma_{\mathsf{F}}$. The same is true for the (l-G) rule.

As some tests have shown, this can make proof search a little bit faster. Thus, these optimization is included in the updated sequent calculus below.

11.8.2 History

Because formulas and theories in the logic of likelihood are finite, the only reason why we get an infinite proof are loops. Thus, to make sure our calculus terminates in any case, we need to detected and prevent loops. Each time a rule of the calculus is to be applied we check if the current sequent was already used once for the same rule. If this is the case, then there is no use in applying the rule again, because we cannot achieve a result different from that obtained by the first application of the rule. Thus, we don't allow the application of such a rule.

The most straightforward way to make a loop check is to store the complete conclusion of each rule, along with its main formula into a history. Each time a rule of the calculus is to be applied, we first check if the main formula of the rule, along with all its side formulas are in the history. If they are, we know that we'd created a loop and thus skip the rule.

This method can be greatly improved, though. A closer look at the rules of the sequent calculus shows that all rules except the K/G-rules decrease the length of the formulas in the premiss. Thus, the only way to get a loop is by using K/G-rules. This means, we don't have to check for a loop each time a rule is used, only when applying a K/G-rule.

When storing the information of the history, it is necessary to store all elements of the sequent that are required for recognizing the loop. While all the elements of the conclusion of a rule are

[5] that $A \in \Sigma_{\mathsf{F}}$ is given from the assumption; lemma 21 then gives us the second assumption $A \mathrel{\dot\in} \Gamma; \Delta$ that is used for lemma 22.

surely sufficient, it is enough to store the elements of the premiss. Although the elements of the premiss don't uniquely identify the sequent before the rule application, a loop is also achieved if the premiss is the same as one already encountered. A look at the K/G-rules shows that the premisses of these rules contain much less formulas and thus require much shorter histories.

In the sequent calculus with loop check, several symbols are used to store the histories. We use the symbols H_{L} to store the history information for L-formulas as main formula. Likewise, H_{K} stores the history information of K-formulas, H_{G} those of G-formulas, and H_{F} those of F-formulas.

A short inspection of the rules of the sequent calculus reveals, that the sets for marked G and F-formulas never diminish. A formula that is present in such a set will never again be removed. If a new formula not present in one of the sets is added, then we can never have a loop, because the set is now bigger than the sets of all previous steps. Thus, it is not necessary to store these sets along with the main formula of the rule. Instead, the main formula alone is stored and each time a new G or F-formula is added to the sets of marked formulas, this storage is cleared. This way, the same loops are recognized as when storing all data, but much less formulas need to be stored.

For this reason, the histories H_{G} and H_{F} are simple sets containing the main formulas of the rule applications. These histories are cleared, whenever a new G or F-formula is marked.

The same is only partially true for the histories Π_{K} and Σ_{L}. While storing the marked G and F-formulas is not necessary as well, we need to store the marked L and K-formulas. The K/G-rules reset the Π_{K} and Σ_{L} sets and thus diminish them. Using the same trick as above is thus not possible.

The histories H_{L} and H_{K} thus have a more complicated structure. Each element of the history set consists of a sequent, itself consisting of the main formula and the corresponding set of marked K and L-formulas. We write this using the notation $(A;\Pi_{\mathsf{K}};\Sigma_{\mathsf{L}})$. Further we use $\{(A;\Pi_{\mathsf{K}};\Sigma_{\mathsf{L}}) \mid A \in \Gamma_{\mathsf{L}}\}$ for the set consisting of the sequents with each formula from Γ_{L} as the main formula, each time with the same set of marked formulas. Just as the other history sets, H_{L} and H_{K} have to be cleared whenever a new F or G-formula is marked.

All together, the different histories have the following structure:

$\mathrm{H}_{\mathsf{L}} =$ $\{A_1, \dots, A_n\}$, where each A_i is a triple $(B_i;\Pi_i;\Sigma_i)$, consisting of a formula B_i, and two multisets Π_i and Σ_i.
$\mathrm{H}_{\mathsf{F}} =$ $\{A_1, \dots, A_n\}$, where each A_i is a formula.
$\mathrm{H}_{\mathsf{K}} =$ $\{A_1, \dots, A_n\}$, where each A_i is a triple $(B_i;\Pi_i;\Sigma_i)$, consisting of a formula B_i, and two multisets Π_i and Σ_i.
$\mathrm{H}_{\mathsf{G}} =$ $\{A_1, \dots, A_n\}$, where each A_i is a formula.

A last optimization step can be done by not only adding the main formula of the rule to the history, but also all other candidates for this rule as well. Because we have to try all these candidates with backtracking if one fails, we don't need to use them while trying the first. This can drastically reduce the amount of time needed for computation.

All these optimizations are included in the sequent calculus that is shown on the next pages.

$$\frac{}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A, \Gamma \supset A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}\ (\text{id}) \qquad \frac{}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \mathsf{false}, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}\ (\text{id-f}) \qquad \frac{}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \mathsf{true}, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}\ (\text{id-t})$$

$$\frac{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \neg A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}\ (\text{l-}\neg) \qquad \frac{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \neg A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}\ (\text{r-}\neg)$$

$$\frac{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}} \quad \mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A \vee B, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}\ (\text{l-}\vee) \qquad \frac{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A, B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A \vee B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}\ (\text{r-}\vee)$$

$$\frac{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A, B, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A \wedge B, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}\ (\text{l-}\wedge) \qquad \frac{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}} \quad \mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A \wedge B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}\ (\text{r-}\wedge)$$

$$\frac{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}} \quad \mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A \rightarrow B, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}\ (\text{l-}\rightarrow) \qquad \frac{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A, \Gamma \supset B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A \rightarrow B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}\ (\text{r-}\rightarrow)$$

$$\frac{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A, B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}} \quad \mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A, B, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A \leftrightarrow B, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}\ (\text{l-}\leftrightarrow)$$

$$\frac{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid A, \Gamma \supset B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}} \quad \mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid B, \Gamma \supset A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A \leftrightarrow B, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}\ (\text{r-}\leftrightarrow)$$

$$\frac{\{(B;\Pi_{\mathsf{K}};\Sigma_{\mathsf{L}}) \mid B \in \{\mathsf{L}A\} \cup \Gamma_{\mathsf{L}}\} \cup \mathrm{H}_{\mathsf{L}}, \Gamma_{\mathsf{F}} \cup \mathrm{H}_{\mathsf{F}} \parallel \emptyset; \Pi_{\mathsf{G}} \mid A, \Pi_{\mathsf{K}}, \Pi_{\mathsf{G}} \supset \Sigma_{\mathsf{L}}, \Sigma_{\mathsf{F}} \mid \emptyset; \Sigma_{\mathsf{F}} \parallel \{(B;\Pi_{\mathsf{K}};\Sigma_{\mathsf{L}}) \mid B \in \Delta_{\mathsf{K}}\} \cup \mathrm{H}_{\mathsf{K}}, \Delta_{\mathsf{G}} \cup \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \mathsf{L}A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}} \; (\text{l-L}, (\mathsf{L}A;\Pi_{\mathsf{K}};\Sigma_{\mathsf{L}}) \notin \mathrm{H}_{\mathsf{L}})$$

$$\frac{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A, \Delta \mid \Sigma_{\mathsf{L}} \cup A; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \mathsf{L}A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}} \; (\text{r-L})$$

$$\frac{\{(B;\Pi_{\mathsf{K}};\Sigma_{\mathsf{L}}) \mid B \in \Gamma_{\mathsf{L}}\} \cup \mathrm{H}_{\mathsf{L}}, \{A,\} \cup \Gamma_{\mathsf{F}} \cup \mathrm{H}_{\mathsf{F}} \parallel \emptyset; \Pi_{\mathsf{G}} \mid A, \Pi_{\mathsf{G}} \supset \Sigma_{\mathsf{F}} \mid \emptyset; \Sigma_{\mathsf{F}} \parallel \{(B;\Pi_{\mathsf{K}};\Sigma_{\mathsf{L}}) \mid B \in \Delta_{\mathsf{K}}\} \cup \mathrm{H}_{\mathsf{K}}, \Gamma_{\mathsf{G}} \cup \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \mathsf{F}A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}} \; (\text{l-F}, A \notin \mathrm{H}_{\mathsf{F}})$$

$$\frac{\emptyset, \emptyset \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \cup A \parallel \emptyset, \emptyset}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \mathsf{F}A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}} \; (\text{r-F}, A \notin \Sigma_{\mathsf{F}})$$

$$\frac{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}} \cup A; \Pi_{\mathsf{G}} \mid A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \mathsf{K}A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}} \; (\text{l-K})$$

$$\frac{\{(B;\Pi_{\mathsf{K}};\Sigma_{\mathsf{L}}) \mid B \in \Gamma_{\mathsf{L}}\} \cup \mathrm{H}_{\mathsf{K}}, \Gamma_{\mathsf{G}} \cup \mathrm{H}_{\mathsf{G}} \parallel \emptyset; \Pi_{\mathsf{G}} \mid \Pi_{\mathsf{K}}, \Pi_{\mathsf{G}} \supset A, \Sigma_{\mathsf{L}}, \Sigma_{\mathsf{F}} \mid \emptyset; \emptyset \parallel \{(B;\Pi_{\mathsf{K}};\Sigma_{\mathsf{L}}) \mid B \in \{\mathsf{K}A\} \cup \Delta_{\mathsf{K}}\} \cup \mathrm{H}_{\mathsf{K}}, \Delta_{\mathsf{F}} \cup \mathrm{H}_{\mathsf{F}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \mathsf{K}A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}} \; (\text{r-K}, (\mathsf{K}A;\Pi_{\mathsf{K}};\Sigma_{\mathsf{L}}) \notin \mathrm{H}_{\mathsf{K}})$$

$$\frac{\emptyset, \emptyset \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \cup A \mid A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \emptyset, \emptyset}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \mathsf{G}A, \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}} \; (\text{l-G}, A \notin \Pi_{\mathsf{G}})$$

$$\frac{\{(B;\Pi_{\mathsf{K}};\Sigma_{\mathsf{L}}) \mid B \in \Gamma_{\mathsf{L}}\} \cup \mathrm{H}_{\mathsf{L}}, \Gamma_{\mathsf{F}} \cup \mathrm{H}_{\mathsf{F}} \parallel \emptyset; \Pi_{\mathsf{G}} \mid \Pi_{\mathsf{G}} \supset A, \Sigma_{\mathsf{F}} \mid \emptyset; \Sigma_{\mathsf{F}} \parallel \{(B;\Pi_{\mathsf{K}};\Sigma_{\mathsf{L}}) \mid B \in \Delta_{\mathsf{K}}\} \cup \mathrm{H}_{\mathsf{K}}, \{A\} \cup \Gamma_{\mathsf{G}} \cup \mathrm{H}_{\mathsf{G}}}{\mathrm{H}_{\mathsf{L}}, \mathrm{H}_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \mathsf{G}A, \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel \mathrm{H}_{\mathsf{K}}, \mathrm{H}_{\mathsf{G}}} \; (\text{r-G}, A \notin \mathrm{H}_{\mathsf{G}})$$

11.8.3 Correctness

Compared to the earlier version of the sequent calculus, all rules just got some additional sets that are stored and additional restrictions when the rule can be used. This does in no way interfere with correctness. Therefore correctness follows from the correctness of the earlier sequent calculus.

Especially, a proof in the sequent calculus with history can easily be transformed into a proof of the sequent calculus without history by simply removing all history sets in all rules.

11.8.4 Completeness

Contrary to correctness, completeness has to be shown again, because the additional restrictions on the rules could prevent some of the proofs used earlier.

Theorem 10 ***Completeness***

$$\vdash_{S} \emptyset;\emptyset \mid \Gamma \supset \Delta \mid \emptyset;\emptyset \quad \Rightarrow \quad \vdash_{S_H} \emptyset \,\|\, \emptyset;\emptyset \mid \Gamma \supset \Delta \mid \emptyset;\emptyset \,\|\, \emptyset$$

Proof
We prove completeness by transforming a proof of the previous sequent calculus into one of the new calculus.

If we have a proof of a sequent in the original sequent calculus that does not contain any loops, we can use completely the same proof in the new calculus and just have to add the histories.

If the proof contains one or more loops, we simply remove all rule applications between the two equal sequents. The resulting structure is still a proof, but does contain one loop less. By doing this for all loops[6], we get a proof with no loops and can easily transform it into a proof in the new calculus as before.

11.8.5 Termination

Lemma 23 ***History Size***
If $\vdash_{S} \emptyset;\emptyset \mid \;\supset A \mid \emptyset;\emptyset$, *then*

$$|H_{\mathsf{L}}| + |H_{\mathsf{F}}| + |H_{\mathsf{K}}| + |H_{\mathsf{G}}| < |A|$$

for each rule of the proof.

[6] a proof has to be finite, thus we can only have a finite number of loops.

Proof
Clearly, the double-sided sequent calculus with history has the subformula property. Thus, every formula appearing in the proof has to be a subformula of A. This enforces that every formula in the sets H_{L}, H_{F}, H_{K}, and H_{G} is a subformula of A. Because the sets H_{F} and H_{G} only contain formulas that stem from different subformulas of A and the sets H_{L} and H_{K} only have L- and K-formulas as the main formulas of their sequents, all sets together cannot have more elements as there are subformulas of A. Thus we have $|\mathrm{H}_{\mathsf{L}}| + |\mathrm{H}_{\mathsf{F}}| + |\mathrm{H}_{\mathsf{K}}| + |\mathrm{H}_{\mathsf{G}}| \leq |A|$.

On the other hand, A must contain at least one propositional variable, required from the definition of the structure of a formula (definition 2). But a propositional variable can never be part of one of the histories, because only L, F, K, and G formulas are stored in histories. Therefore, there is at least one subformula of A which is not present in the history sets, thus the lemma holds.

Lemma 24 ***Marked Formula Size***
If $\vdash_S \emptyset;\emptyset \mid \ \supset A \mid \emptyset;\emptyset$, *then*

$$|\Pi_{\mathsf{K}}| + |\Pi_{\mathsf{G}}| + |\Sigma_{\mathsf{L}}| + |\Sigma_{\mathsf{F}}| < |A|$$

for each rule of the proof.

Proof
The proof is the similar to the one of the previous lemma. Every formula of the Π and Σ sets must be a subformula of A and all these formulas were created from a different subformulas of A. Furthermore, A itself cannot be part of any of the formulas, because the top level operator of the formula is removed when it is put into one of the multisets.

These lemmas make sure that $|A| - (|\mathrm{H}_{\mathsf{L}}| + |\mathrm{H}_{\mathsf{F}}| + |\mathrm{H}_{\mathsf{K}}| + |\mathrm{H}_{\mathsf{G}}|) > 0$ and $|A| - (|\Pi_{\mathsf{K}}| + |\Pi_{\mathsf{G}}| + |\Sigma_{\mathsf{L}}| + |\Sigma_{\mathsf{F}}| > 0$. This information is used below.

Definition 29 ***Sequent Measure***
We assume that we have a proof of the formula A. We define the following measure for sequents *of a proof of the calculus:*

$$m(H_{\mathsf{L}}, H_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel H_{\mathsf{K}}, H_{\mathsf{G}}) = \\ (|\Gamma| + |\Delta| + |A|) \cdot (|A| - (|H_{\mathsf{L}}| + |H_{\mathsf{F}}| + |H_{\mathsf{K}}| + |H_{\mathsf{G}}|)$$

This definition ensures that we have $m(\ldots) \geq 0$ for every sequent that appears in the proof of A.

Lemma 25 ***Termination***
If $H_{\mathsf{L}}, H_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel H_{\mathsf{K}}, H_{\mathsf{G}}$ is the conclusion in the a proof of A and furthermore the sequent $H'_{\mathsf{L}}, H'_{\mathsf{F}} \parallel \Pi'_{\mathsf{L}}; \Pi'_{\mathsf{F}} \mid \Gamma' \supset \Delta' \mid \Sigma'_{\mathsf{L}}; \Sigma'_{\mathsf{F}} \parallel H'_{\mathsf{K}}, H'_{\mathsf{G}}$ one of its premisses, then we have

$$m(H_{\mathsf{L}}, H_{\mathsf{F}} \parallel \Pi_{\mathsf{K}}; \Pi_{\mathsf{G}} \mid \Gamma \supset \Delta \mid \Sigma_{\mathsf{L}}; \Sigma_{\mathsf{F}} \parallel H_{\mathsf{K}}, H_{\mathsf{G}}) < \\ m(H'_{\mathsf{L}}, H'_{\mathsf{F}} \parallel \Pi'_{\mathsf{L}}; \Pi'_{\mathsf{F}} \mid \Gamma' \supset \Delta' \mid \Sigma'_{\mathsf{L}}; \Sigma'_{\mathsf{F}} \parallel H'_{\mathsf{K}}, H'_{\mathsf{G}})$$

Proof
We distinguish different cases according to the rule in question (we don't have to treat axioms, because they don't have premisses):

(l-¬) According to the rule, we move a formula from left to the right side and remove the negation operator. Thus, the total length of Γ and Δ surely is reduced, i.e. $|\Gamma| + |\Delta| < |\Gamma'| + |\Delta'|$, while the rest of the sequent stays the same. Thus, the proposition holds.

(r-¬) This can be done analogously to the previous case.

(l-∨) Only one part of the conjunction is present in the premiss treated, thus Γ is surely reduced, while all the rest of the sequent stays the same. Thus, m decreases.

(r-∨) As with the previous rules, this rule does remove one operator from the formulas of Δ and leaves the rest untouched. Thus, m surely is reduced.

(l-∧) This is dual to the case for (r-∨).

(r-∧) This is dual to the case for (l-∨).

(l-→) This is dual to the case for (l-∨).

(r-→) This is dual to the case for (r-∨).

(l-↔) Although both sides of the equivalence operator are taken over into the premiss, the operator itself is not, thus the length is still reduced. All other parts of the sequent again stay the same.

(r-↔) This is the same as for (l-↔).

(l-L) In this case, it is not necessary that $|\Gamma| + |\Delta|$ is reduced. It is quite possible that the length of these formulas increases drastically. The maximal amount these length can increase is $|\mathrm{H}_{\mathsf{L}}| + |\mathrm{H}_{\mathsf{F}}| + |\mathrm{H}_{\mathsf{K}}| + |\mathrm{H}_{\mathsf{G}}| - 1$, i.e. if we only have a formula $\mathsf{L}A$ in the conclusion. But according to lemma 24 this is smaller than $|A|$.

At the same time, the number of formulas stored in the history is increased by the rule at least by one. Thus, $|A| - |H|$ is reduced by at least one, and therefore m is reduced at least by $|A|$. At the same time, it's increased by at least $|A| - 1$, thus overall m is at least decreased by one.

(r-L) This is the same as for (r-∨).

… The rest of the rules can be proven similarly.

With the previous lemma, it is quite clear that backward proof search must always terminate. At the beginning, m is finite and for each rule in the proof it is reduce by at least 1. Thus, after a finite number of steps the proof has to be finished.

11.9 Embeddings

We have already seen in chapter 8 that the logic of likelihood LL^- has some similarities to modal logics, mainly to KT and S_4. In this section, we show that the logics KT and S_4 can easily be embedded in the logic of likelihood. We will use these embeddings to use the benchmark formulas for KT and S_4 of [2] to test the efficiency of the implementation of the prover for LL^-.

Definition 30 ***Transformation from*** S_4
We define the transformation $\star$ of a formula A of S_4 to a formula $A^\star \in \mathrm{Fml}_{\mathrm{LL}^-}$ by:

- $p_i^\star = p_i$,
- $(\neg A)^\star = \neg A^\star$,
- $(A \vee B)^\star = A^\star \vee B^\star$,
- $(\Box A)^\star = \mathsf{G}A^\star$,
- $(\Diamond A)^\star = \mathsf{F}A^\star$.

Definition 31 ***Transformation from*** KT
We define the transformation $*$ of a formula A of KT to a formula $A^* \in \mathrm{Fml}_{\mathrm{LL}^-}$ by:

- $p_i^* = p_i$,
- $(\neg A)^* = \neg A^*$,
- $(A \vee B)^* = A^* \vee B^*$,
- $(\Box A)^* = \mathsf{K}A^*$,
- $(\Diamond A)^* = \mathsf{L}A^*$.

Lemma 26 ***Embedding of*** S_4

$$\vdash_{S4} A \quad \Rightarrow \quad \vdash_{H} A^\star$$

Proof
We have to prove that all axioms and inference rules of S_4 are provable in the Hilbert calculus for LL^-:

classical:	surely all valid classical formulas are provable, because they are axioms of the Hilbert calculus of LL^-.
$\Box(A \to B) \to (\Box A \to \Box B)$:	the translation of this is $\mathsf{G}(A^\star \to B^\star) \to (\mathsf{G}A^\star \to \mathsf{G}B^\star)$, which is exactly axiom (AX7).
$\Box A \to A$:	the translation of this is $\mathsf{G}A^\star \to A^\star$, which is axiom (AX2).
$\Box A \to \Box\Box A$:	the translation of this is $\mathsf{G}A^\star \to \mathsf{GG}A^\star$, which is axiom (AX3).
modus ponens:	the same inference rule is used in the Hilbert calculus for LL^-.

generalization:	this generalization is the same as (R1).

Lemma 27 ***Embedding of*** KT

$$\vdash_{\overline{KT}} A \quad \Rightarrow \quad \vdash_{\overline{H}} A^*$$

Proof
We have to prove that all axioms and inference rules of KT are provable in the Hilbert calculus for LL$^-$:

classical:	surely all valid classical formulas are provable, because they are axioms of the Hilbert calculus of LL$^-$.
$\Box(A \to B) \to (\Box A \to \Box B)$:	the translation of this is $\mathsf{K}(A^* \to B^*) \to (\mathsf{K}A^* \to \mathsf{K}B^*)$, which can be easily obtained through (AX6).
$\Box A \to A$:	the translation of this is $\mathsf{K}A \to A$, i.e. $\neg A \to \mathsf{L}\neg A$. This is axiom (AX5).
modus ponens:	the same inference rule is used in the Hilbert calculus for LL$^-$.
generalization:	this generalization can be easily proven by (R1) and the fact that $\mathsf{G}A \to \mathsf{K}A$ (AX4).

Chapter 12

LWB Example

This chapter contains an extensive example that shows how the logic of likelihood can be used. The example is taken from the original paper [22] by Rabin and Halpern. The example is not only repeated here, but it is also verified using the Logics Workbench. To do that, it uses the implementation described in the next chapter.

The example deals with the verification and proving of properties of a protocol for the exchange of secrets. After an introduction into the protocol and some notational remarks, the protocol is described in detail. Afterwards, a simplified example will be proven, followed by a proof of some properties of the protocol. All proofs were done using the LWB and the statements used for the proofs are always shown as well.

12.1 Introduction

The example in this chapter verifies a protocol for data transmission. It proves several properties of the protocol and can also generally analyze it. For example, proofs are given to show that cheating with the protocol is not possible or at least not advantageous.

As it is tradition in cryptography, the two participants in the data transmission are called Alice and Bob. Alice and Bob would like to transmit one bit of secret information to each other. We can assume that the secrets transmitted are passwords to specific files[1], a file for Alice and one for Bob.

The protocol is meant to be self-enforcing, i.e. there is no third party which has to be trusted to adjudicate in disputes. To achieve this, the files are assumed to be trapped in a way that—if a file is opened with the wrong password—*both* files are destroyed. This prevents Bob and Alice from guessing the password and, under certain circumstances detailed below, also prevents cheating. Furthermore, we assume that Bob can tell when Alice opens his file and vice versa.

[1] The transmission and the protocol shown below are only done for a single bit. If multiple bits are to be transferred, then the same techniques may be used several times to get the desired result.

12.1.1 Notation

We call the secret of Alice, i.e. the password to open Alice's file S_A. Dually, the secret of Bob is called S_B. In the following, information of Alice will always use the index A, while information of Bob uses the index B.

The protocol uses addition modulo 2, denoted by $\oplus$.

12.2 Protocol

12.2.1 Oblivious Transfer

Before we actually go into the details of the protocol used to transmit information, we have to take a look at a special type of communication line used to transfer part of the information of the protocol. While most of the information transmitted is sent over regular communication lines, some crucial information has to be sent specially. For that reason, the oblivious transfer was developed.

A transmission using oblivious transfer has the following properties:

- the probability is exactly 50% that the message sent is really transmitted,
- the recipient knows if he got the message or not,
- the sender does not know if the recipient got the message or not.

The properties of the transfer sound quite strange and it's actually not easy to see that such a transformation can actually be implemented. In [3], for example, a practical method is shown how to implement oblivious transfer using faint pulses of polarized light. We won't go into further details how this sort of transformation is actually done, because the only things important are the properties mentioned above.

12.2.2 Steps of the Protocol

The protocol proceeds along the following four steps:

Step 1. (a) Alice sends a random bit (R_A) to Bob, using an oblivious transfer,

(b) Bob sends a random bit (R_B) to Alice, using an oblivious transfer.

Step 2. (a) Alice sets $\mu_A = 1$ if Bob's random bit was transferred to her successfully, and $\mu_A = 0$ else,

(b) Bob does the same with μ_B and the message he got from Alice.

Step 3. (a) Alice sends $S_A \oplus \mu_A$ to Bob, using a normal communication line,

(b) Bob sends $S_B \oplus \mu_B$ to Alice, using a normal communication line.

Step 4. (a) Alice sends $S_A \oplus R_A$ to Bob, again using a normal communication line,

(b) Bob sends $S_B \oplus R_B$ to Alice, also using a normal communication line.

After these four steps are carried through, both Alice and Bob may be capable of determining the other's secret. After step three, Bob is not yet capable to determine Alice's secret R_A, because he has no way of determining μ_A, which depends on the oblivious transfer. But if the oblivious transfer form Bob to Alice transmitted R_A, he can determine S_A after step 4. The same holds true for Alice. If, on the other hand, Bob's transfer worked while Alice's did not, Alice can directly determine R_B, but Bob can not. If now Alice opens Bob's file, he knows that his transfer must have worked, allowing him to deduce $\mu_A = 1$. This information, together with the information from step 3 allows him to compute S_A as well. Thus, either both are able to deduce the secrets, or in the case of both transfers not working, both don't know the other's secret. Thus, the protocol gives exactly 75% chance that the secrets are exchanged.

12.2.3 Simplified Example

Example 15 ***Simplified Example***

Before we look at the detailed example, we have a look at a simplified version, as it is done in [22].

In this example, we want to see what would happen if Bob lies about his password to Alice. To further simplify, we assume that the password to Bob's file is 0, i.e. $S_B = 0$.

This simplified version just uses the following propositional variables to express properties of the protocol[2]:

$(\mathsf{B}, \mathsf{T}, 1)$	**B**ob **t**ells Alice that his password is **1**,
$(\mathsf{A}, \mathsf{E}, 1)$	**A**lice **e**nters Bob's file with password **1**,
(DS)	both files are **des**troyed.

We additionally write down some extra logical axioms that define properties of the protocol we want to take for granted. For this short example, the following three axioms (or hypotheses) are enough:

(a) $\mathsf{G}(\mathsf{A}, \mathsf{E}, 1) \to \mathsf{G}(\mathsf{DS})$,

(b) $\mathsf{G}\neg(\mathsf{DS})$,

(c) $\mathsf{G}(\mathsf{B}, \mathsf{T}, 1) \to \mathsf{L}^k\mathsf{G}(\mathsf{A}, \mathsf{E}, 1)$ for some k.

[2] for better reading we don't use the p_{X} notation here.

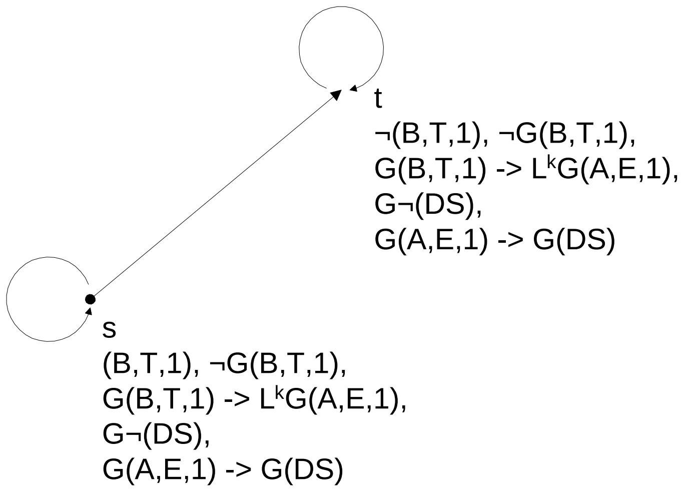


Figure 12.1: Countermodel for $\mathsf{G}\neg(\mathsf{B},\mathsf{T},1)$

The first axiom just states that the password of Bob is not 1, i.e. if Alice uses that password then this will destroy both files. The second axiom says that nobody wants that the files are destroyed. The last axiom expresses, that if Bob transmits to Alice that his password is 1, then Alice will likely use that password to open Bob's file. This is an assumption of credibility. How much trust Alice puts into the information she got from Bob can be expressed with the given k. The higher k the more unlikely it is that Alice uses the password and thus the less trust Alice gives to the information obtained by Bob.

From these axiom we can now prove $\neg\mathsf{G}(\mathsf{B},\mathsf{T},1)$.[3]

We now check these results with the theorem prover for LL$^-$ in the LWB. First we have to define a theory with the extra logical axioms mentioned above. This can be written in the LWB syntax like this:

```
    theory := [ G(AE1) -> G(DS),
                G(~DS),
                G(BT1) -> LG(AE1) ];
```

[3] in [22] the statement is said to be $\mathsf{G}\neg(\mathsf{B},\mathsf{T},1)$, which is not provable. Consider the counter example given in picture 12.1 or the output of the LWB prover for LL$^-$. Actually, $\mathsf{G}\neg(\mathsf{B},\mathsf{T},1)$ is the same $\neg(\mathsf{B},\mathsf{T},1)$ and could be translated as "we take the hypothesis that Bob won't tell Alice the password is 1 for now", whereas $\neg\mathsf{G}(\mathsf{B},\mathsf{T},1)$ says that Bob actually does not send the wrong password in any case. The latter surely is the statement that is interesting to be proven.

Now we can use the `provable` function to show that our assumption can be proven. We can also show that the original assumption from [22] cannot be proven:

```
      llh> provable(G(~(BT1)), theory);
               false
      llh> provable(~(BT1), theory);
               false
      llh> provable(~G(BT1), theory);
               true
```

The statement from [22] that—if the second of the extra-logical axioms above is weakened, then only weakened version of our assumption can be proven—does not hold. Even if we replace the second axiom by something weaker, like

$$\neg \mathsf{L}^n \mathsf{G}(\mathsf{DS})$$

it is still possible to prove our assumption. The reason here is that our adjusted assumption is actually much weaker than the—not provable—assumption from the paper.

Negations

Another reason why weakening the theory by adding L-operators does not change the results, lies in the fact that the logic of likelihood does not actually show the expected properties for negated formulas. If we interpret $\neg\mathsf{L}A$ as meaning "it's not likely that A" then we would think that from this we cannot deduce $\neg A$. But this deduction step is nothing else than the contraposition of AX5. On the other hand, we'd also think that from $\neg\mathsf{L}A$ it is not possible to deduce $\neg\mathsf{L}\mathsf{L}A$. But again, this is possible. This can be proven using the LWB, for example:

```
      llh> provable(~LLA, [~LA]);
               true
```

While in the first case, the implication can be proven, the second case actually needs the fact that the formula is provable (i.e. uses generalization). A proof in the Hilbert calculus is just a little more complex:

$\vdash_{\overline{H}}$	$\neg\mathsf{L}A$	(assumption)
$\vdash_{\overline{H}}$	$\mathsf{G}\neg\mathsf{L}A$	(R1)
$\vdash_{\overline{H}}$	$\mathsf{G}\neg\mathsf{L}A \rightarrow \neg\mathsf{L}\mathsf{L}A$	(AX4)
$\vdash_{\overline{H}}$	$\neg\mathsf{L}\mathsf{L}A$	(R2 with the last two lines)

On the other hand, the implication

$$\neg \mathsf{L}A \rightarrow \neg \mathsf{L}\mathsf{L}A$$

cannot be proven in LL$^-$.

Summarized, if we have a negation in front of a formula, then the number of L-operators is irrelevant. This does actually contradict the understanding one has of a logic of likelihood. Thus, it is best only to treat non-negated formulas.

12.3 Detailed Example

Example 16 ***Detailed Example***

After the simplified example above, we can now treat the protocol in all its details. For the detailed example we need much more propositional variables.

From the view of Alice, we need the following variables:

$(S_A, 0)$	the value of the password of Alice is 0,
$(S_A, 1)$	the value of the password of Alice is 1,
$(\mu_A, 0)$	the oblivious transfer from Bob did not succeed,
$(\mu_A, 1)$	the oblivious transfer from Bob did succeed,
$(\mathsf{A}, \mathsf{S}, R_A, 0)$	Alice sends as random bit the value 0,
$(\mathsf{A}, \mathsf{S}, R_A, 1)$	Alice sends as random bit the value 1,
$(\mathsf{A}, \mathsf{S}, S_A \oplus \mu_A, 0)$	Alice sends for the addition of S_A and μ_A the value 0,
$(\mathsf{A}, \mathsf{S}, S_A \oplus \mu_A, 1)$	Alice sends for the addition of S_A and μ_A the value 1,
$(\mathsf{A}, \mathsf{S}, S_A \oplus R_A, 0)$	Alice sends for the addition of S_A and R_A the value 0,
$(\mathsf{A}, \mathsf{S}, S_A \oplus R_A, 1)$	Alice sends for the addition of S_A and R_A the value 1,
$(\mathsf{A}, \mathsf{D}, R_B, 0)$	Alice deduces that Bob's random value is 0,
$(\mathsf{A}, \mathsf{D}, R_B, 1)$	Alice deduces that Bob's random value is 1,
$(\mathsf{A}, \mathsf{D}, S_B, 0)$	Alice deduces that Bob's password is 0,
$(\mathsf{A}, \mathsf{D}, S_B, 1)$	Alice deduces that Bob's password is 1,
$(\mathsf{A}, \mathsf{D}, \mu_B, 0)$	Alice deduces that her oblivious transfer to Bob did not succeed,
$(\mathsf{A}, \mathsf{D}, \mu_B, 1)$	Alice deduces that her oblivious transfer to Bob did succeed,
$(\mathsf{A}, \mathsf{D}, S_B \oplus \mu_B, 0)$	Alice deduces that Bob has the value 0 for the addition of S_B and μ_B,
$(\mathsf{A}, \mathsf{D}, S_B \oplus \mu_B, 1)$	Alice deduces that Bob has the value 1 for the addition of S_B and μ_B,
$(\mathsf{A}, \mathsf{D}, S_B \oplus R_B, 0)$	Alice deduces that the value of the addition of Bob's S_B and R_B is 0,
$(\mathsf{A}, \mathsf{D}, S_B \oplus R_B, 1)$	Alice deduces that the value of the addition of Bob's S_B and R_B is 1,
$(\mathsf{A}, \mathsf{E}, 0)$	Alice enters Bob's file with password 0,
$(\mathsf{A}, \mathsf{E}, 1)$	Alice enters Bob's file with password 1,
(DS)	both files are destroyed.

We need a similar set of variables—with interchanged roles—for the viewpoint of Bob, with the exception of (DS).

Now that we have the necessary variables, we can list the extra-logical axioms necessary for Alice:

(a) $\neg(\mathsf{G}(S_A,0) \wedge \mathsf{G}(S_A,1))$

(b) $\neg(\mathsf{G}(\mu_A,0) \wedge \mathsf{G}(\mu_A,1))$

(c) $\mathsf{G}(\mathsf{A},\mathsf{S},R_A,0) \rightarrow \mathsf{LG}(\mathsf{B},\mathsf{D},R_A,0)$
$\mathsf{G}(\mathsf{A},\mathsf{S},R_A,1) \rightarrow \mathsf{LG}(\mathsf{B},\mathsf{D},R_A,1)$

(d) $\mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus \mu_A,0) \leftrightarrow \mathsf{G}(\mathsf{B},\mathsf{D},S_A \oplus \mu_A,0) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus R_A,0) \leftrightarrow \mathsf{G}(\mathsf{B},\mathsf{D},S_A \oplus R_A,0)$
$\mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus \mu_A,1) \leftrightarrow \mathsf{G}(\mathsf{B},\mathsf{D},S_A \oplus \mu_A,1) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus R_A,1) \leftrightarrow \mathsf{G}(\mathsf{B},\mathsf{D},S_A \oplus R_A,1)$

(e) $[(\mathsf{G}(\mathsf{B},\mathsf{D},S_A \oplus R_A,0) \wedge \mathsf{G}(\mathsf{B},\mathsf{D},R_A,0) \vee (\mathsf{G}(\mathsf{B},\mathsf{D},S_A \oplus \mu_A,0) \wedge \mathsf{G}(\mathsf{B},\mathsf{D},\mu_A,0))] \rightarrow \mathsf{G}(\mathsf{B},\mathsf{D},S_A,0)$
$[(\mathsf{G}(\mathsf{B},\mathsf{D},S_A \oplus R_A,0) \wedge \mathsf{G}(\mathsf{B},\mathsf{D},R_A,1) \vee (\mathsf{G}(\mathsf{B},\mathsf{D},S_A \oplus \mu_A,0) \wedge \mathsf{G}(\mathsf{B},\mathsf{D},\mu_A,1))] \rightarrow \mathsf{G}(\mathsf{B},\mathsf{D},S_A,1)$
$[(\mathsf{G}(\mathsf{B},\mathsf{D},S_A \oplus R_A,1) \wedge \mathsf{G}(\mathsf{B},\mathsf{D},R_A,0) \vee (\mathsf{G}(\mathsf{B},\mathsf{D},S_A \oplus \mu_A,1) \wedge \mathsf{G}(\mathsf{B},\mathsf{D},\mu_A,0))] \rightarrow \mathsf{G}(\mathsf{B},\mathsf{D},S_A,1)$
$[(\mathsf{G}(\mathsf{B},\mathsf{D},S_A \oplus R_A,1) \wedge \mathsf{G}(\mathsf{B},\mathsf{D},R_A,1) \vee (\mathsf{G}(\mathsf{B},\mathsf{D},S_A \oplus \mu_A,1) \wedge \mathsf{G}(\mathsf{B},\mathsf{D},\mu_A,1))] \rightarrow \mathsf{G}(\mathsf{B},\mathsf{D},S_A,0)$

(f) $[\neg\mathsf{G}(\mathsf{B},\mathsf{E},0) \wedge \neg\mathsf{G}(\mathsf{B},\mathsf{E},1) \wedge (\mathsf{G}(\mathsf{A},\mathsf{E},0) \vee \mathsf{G}(\mathsf{A},\mathsf{E},1))] \rightarrow \mathsf{G}(\mathsf{B},\mathsf{D},\mu_A,1) \wedge \mathsf{G}(\mu_A,1)$

(g) $\mathsf{G}(\mathsf{B},\mathsf{D},S_A,0) \rightarrow \mathsf{LG}(\mathsf{B},\mathsf{E},0)$
$\mathsf{G}(\mathsf{B},\mathsf{D},S_A,1) \rightarrow \mathsf{LG}(\mathsf{B},\mathsf{E},1)$

(h) $(\mathsf{G}(S_A,0) \wedge \mathsf{G}(\mathsf{B},\mathsf{E},1)) \rightarrow (\mathsf{DS})$
$(\mathsf{G}(S_A,1) \wedge \mathsf{G}(\mathsf{B},\mathsf{E},0)) \rightarrow (\mathsf{DS})$

The axioms can be interpreted as follows:

(a) Because we do all computations in a propositional calculus, we need two variables to express that S_A has the value 0 or 1. This axiom makes sure, that these two variables actually represent the value of S_A, i.e. it can have either the value 0 or the value 1 but not never both.

(b) We need to have the same for μ_A as for S_A.

(c) These are actually two axioms, depending on the value Alice sends for R_A. The axiom says that if Alice sends the random bit R_A using oblivious transfer, then it is likely that Bob will be able to deduce the value of R_A. The L-operator expresses that the transmission of R_A is done using oblivious transfer, which only allows Bob to receive the value in 50% of the cases.

(d) These several axioms state that the transmission of $S_A \oplus \mu_A$ and $S_A \oplus R_A$ are done using regular, error free communication lines. This means, what Alice sends will be received by Bob.

(e) The axioms in this group describe how Bob can compute the password of Alice from the information he obtained. If he got the random bit of Alice, then he can deduce S_A from $S_A \oplus R_A$ sent by Alice. If Bob can somehow deduce the if Alice got his oblivious transfer transmission, i.e. μ_A, then he can deduce the password from $S_A \oplus \mu_A$ received from Alice.

(f) This axiom says, that if Alice enters Bob file before he enters hers, i.e. $(\mathsf{B}, \mathsf{E}, 0)$ and $(\mathsf{B}, \mathsf{E}, 1)$ are not true but one of $(\mathsf{A}, \mathsf{E}, 0)$ or $(\mathsf{A}, \mathsf{E}, 1)$ is, then Bob can deduce that Alice must have received his random bit sent by oblivious transfer and thus that $\mu_A = 1$.

(g) As in the short example above, these are the credibility assumptions. They state that if Bob can deduce the password of Alice he will likely trust her and enter her file with that password. Depending on the degree of trust, these axioms could be adjusted to have more L on the right side of the implication.

(h) These last axioms just state that if Bob enters Alice's file with the wrong password, then both files will be destroyed.

The same set of axioms can be written down from the point of view of Bob. Because we only want to prove statements from the point of view of Alice, we don't need this second set of axioms right now.

Before we start proving that cheating is not possible with this protocol, we have to look at the theory we need to use to make the proofs within the Logics Workbench. The following theory is exactly the same as the one above, just written in the LWB syntax:

```
    Alice :=
  [
    ~(G(SA_0) & G(SA_1)),

    ~(G(MuA_0) & G(MuA_1)),

    G(A_S_RA_0) -> LG(B_D_RA_0),
    G(A_S_RA_1) -> LG(B_D_RA_1),

    ((G(A_S_SA_pl_MuA_0) <-> G(B_D_SA_pl_MuA_0)) &
       (G(A_S_SA_pl_RA_0) <-> G(B_D_SA_pl_RA_0)),
    (G(A_S_SA_pl_MuA_1) <-> G(B_D_SA_pl_MuA_1)) &
       (G(A_S_SA_pl_RA_1) <-> G(B_D_SA_pl_RA_1)),

    ((G(B_D_SA_pl_RA_0) & G(B_D_RA_0)) v
       (G(B_D_SA_pl_MuA_0) & G(B_D_MuA_0))) -> G(B_D_SA_0),
    ((G(B_D_SA_pl_RA_0) & G(B_D_RA_1)) v
       (G(B_D_SA_pl_MuA_0) & G(B_D_MuA_1))) -> G(B_D_SA_1),
    ((G(B_D_SA_pl_RA_1) & G(B_D_RA_0)) v
       (G(B_D_SA_pl_MuA_1) & G(B_D_MuA_0))) -> G(B_D_SA_1),
    ((G(B_D_SA_pl_RA_1) & G(B_D_RA_1)) v
       (G(B_D_SA_pl_MuA_1) & G(B_D_MuA_1))) -> G(B_D_SA_0),

    (~G(B_E_0) & ~G(B_E_1) & (G(A_E_0) v G(A_E_1))) ->
       (G(B_D_MuA_1) & G(MuA_1)),

    G(B_D_SA_0) -> LG(B_E_0),
    G(B_D_SA_1) -> LG(B_E_1),
    (G(SA_0) & G(B_E_1)) -> (DS),
    (G(SA_1) & G(B_E_0)) -> (DS)
  ];
```

12.3.1 Cheating in Step 4

If Alice wants to cheat in Step 4, then she can do this only by sending the wrong value for $S_A \oplus R_A$. This is the meaning of the following formula:

$$\begin{aligned}(\mathsf{A},\mathsf{C},4) \equiv\ & (\mathsf{G}(\mathsf{A},\mathsf{S},R_A,0) \wedge \mathsf{G}(S_A,0) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus R_A,1))\vee \\ & (\mathsf{G}(\mathsf{A},\mathsf{S},R_A,0) \wedge \mathsf{G}(S_A,1) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus R_A,0))\vee \\ & (\mathsf{G}(\mathsf{A},\mathsf{S},R_A,1) \wedge \mathsf{G}(S_A,0) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus R_A,0))\vee \\ & (\mathsf{G}(\mathsf{A},\mathsf{S},R_A,1) \wedge \mathsf{G}(S_A,1) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus R_A,1))\end{aligned}$$

This is written in the LWB as follows:

```
    A_C_4 := (G(A_S_RA_0) & G(SA_0) & G(A_S_SA_pl_RA_1)) v
           (G(A_S_RA_0) & G(SA_1) & G(A_S_SA_pl_RA_0)) v
           (G(A_S_RA_1) & G(SA_0) & G(A_S_SA_pl_RA_0)) v
           (G(A_S_RA_1) & G(SA_1) & G(A_S_SA_pl_RA_1));
```

Now we can prove in LL^- that—if Alice really cheats in step 4—it is somewhat likely that both files will be destroyed. This is expressed with

$$(\mathsf{A},\mathsf{C},4) \rightarrow \mathsf{L}^2\mathsf{G}(\mathsf{DS})$$

This is in the LWB

```
    provable(A_C_4 -> LLG(DS), Alice);
           true
```

The contraposition of the above formula gives us $\neg\mathsf{L}^2\mathsf{G}(\mathsf{DS}) \rightarrow \neg(\mathsf{A},\mathsf{C},4)$. Thus, if Alice does not want that its somewhat likely that the files will be destroyed, then she should not cheat in step 4. Of course, we can also prove $(\mathsf{A},\mathsf{C},4) \rightarrow \mathsf{L}^k\mathsf{G}(\mathsf{DS})$ for all $k > 2$. It is not possible to prove $(\mathsf{A},\mathsf{C},4) \rightarrow \mathsf{LG}(\mathsf{DS})$ or even $(\mathsf{A},\mathsf{C},4) \rightarrow \mathsf{G}(\mathsf{DS})$, though.

```
    provable(A_C_4 -> LLLLLG(DS), Alice);
           true
    provable(A_C_4 -> LG(DS), Alice);
           false
    provable(A_C_4 -> G(DS), Alice);
           false
```

This computation takes roughly 13 minutes.[4] While this, by itself, is not really a problem, it would be good, especially for the following computations, if the execution time could be somewhat reduced. Thus, we simplify the theory a bit to increase computation speed.

A first simplification is to replace the last two axioms of the set with

$$(\mathsf{G}(S_A, 0) \land \mathsf{G}(\mathsf{B}, \mathsf{E}, 1)) \to \mathsf{G}(\mathsf{DS})$$
$$(\mathsf{G}(S_A, 1) \land \mathsf{G}(\mathsf{B}, \mathsf{E}, 0)) \to \mathsf{G}(\mathsf{DS}).$$

This is actually even more natural and directly corresponds to the extra-logical axioms used for the simple example above. Furthermore, it is easy to show that with $\vdash \mathsf{G}A \to B$ we can also prove $\vdash \mathsf{G}A \to \mathsf{G}B$ and vice versa. Thus, the simplified theory is equivalent to the original one.

Because we are interested in proveing formulas with $\mathsf{G}(\mathsf{DS})$ instead of just (DS), our new theory does speed up provability quite a bit. The example above takes now, with these changes, just somewhat below one minute, instead of the 13 minutes it took before.

Below, we will use the following, slightly modified theory

(a) $\neg(\mathsf{G}(S_A, 0) \land \mathsf{G}(S_A, 1))$

(b) $\neg(\mathsf{G}(\mu_A, 0) \land \mathsf{G}(\mu_A, 1))$

(c) $\mathsf{G}(\mathsf{A}, \mathsf{S}, R_A, 0) \to \mathsf{LG}(\mathsf{B}, \mathsf{D}, R_A, 0)$
$\mathsf{G}(\mathsf{A}, \mathsf{S}, R_A, 1) \to \mathsf{LG}(\mathsf{B}, \mathsf{D}, R_A, 1)$

(d) $\mathsf{G}(\mathsf{A}, \mathsf{S}, S_A \oplus \mu_A, 0) \leftrightarrow \mathsf{G}(\mathsf{B}, \mathsf{D}, S_A \oplus \mu_A, 0) \land \mathsf{G}(\mathsf{A}, \mathsf{S}, S_A \oplus R_A, 0) \leftrightarrow \mathsf{G}(\mathsf{B}, \mathsf{D}, S_A \oplus R_A, 0)$
$\mathsf{G}(\mathsf{A}, \mathsf{S}, S_A \oplus \mu_A, 1) \leftrightarrow \mathsf{G}(\mathsf{B}, \mathsf{D}, S_A \oplus \mu_A, 1) \land \mathsf{G}(\mathsf{A}, \mathsf{S}, S_A \oplus R_A, 1) \leftrightarrow \mathsf{G}(\mathsf{B}, \mathsf{D}, S_A \oplus R_A, 1)$

(e) $[(\mathsf{G}(\mathsf{B}, \mathsf{D}, S_A \oplus R_A, 0) \land \mathsf{G}(\mathsf{B}, \mathsf{D}, R_A, 0) \lor (\mathsf{G}(\mathsf{B}, \mathsf{D}, S_A \oplus \mu_A, 0) \land \mathsf{G}(\mathsf{B}, \mathsf{D}, \mu_A, 0))] \to \mathsf{G}(\mathsf{B}, \mathsf{D}, S_A, 0)$
$[(\mathsf{G}(\mathsf{B}, \mathsf{D}, S_A \oplus R_A, 0) \land \mathsf{G}(\mathsf{B}, \mathsf{D}, R_A, 1) \lor (\mathsf{G}(\mathsf{B}, \mathsf{D}, S_A \oplus \mu_A, 0) \land \mathsf{G}(\mathsf{B}, \mathsf{D}, \mu_A, 1))] \to \mathsf{G}(\mathsf{B}, \mathsf{D}, S_A, 1)$
$[(\mathsf{G}(\mathsf{B}, \mathsf{D}, S_A \oplus R_A, 1) \land \mathsf{G}(\mathsf{B}, \mathsf{D}, R_A, 0) \lor (\mathsf{G}(\mathsf{B}, \mathsf{D}, S_A \oplus \mu_A, 1) \land \mathsf{G}(\mathsf{B}, \mathsf{D}, \mu_A, 0))] \to \mathsf{G}(\mathsf{B}, \mathsf{D}, S_A, 1)$
$[(\mathsf{G}(\mathsf{B}, \mathsf{D}, S_A \oplus R_A, 1) \land \mathsf{G}(\mathsf{B}, \mathsf{D}, R_A, 1) \lor (\mathsf{G}(\mathsf{B}, \mathsf{D}, S_A \oplus \mu_A, 1) \land \mathsf{G}(\mathsf{B}, \mathsf{D}, \mu_A, 1))] \to \mathsf{G}(\mathsf{B}, \mathsf{D}, S_A, 0)$

(f) $[\neg\mathsf{G}(\mathsf{B}, \mathsf{E}, 0) \land \neg\mathsf{G}(\mathsf{B}, \mathsf{E}, 1) \land (\mathsf{G}(\mathsf{A}, \mathsf{E}, 0) \lor \mathsf{G}(\mathsf{A}, \mathsf{E}, 1))] \to \mathsf{G}(\mathsf{B}, \mathsf{D}, \mu_A, 1) \land \mathsf{G}(\mu_A, 1)$

(g) $\mathsf{G}(\mathsf{B}, \mathsf{D}, S_A, 0) \to \mathsf{LG}(\mathsf{B}, \mathsf{E}, 0)$
$\mathsf{G}(\mathsf{B}, \mathsf{D}, S_A, 1) \to \mathsf{LG}(\mathsf{B}, \mathsf{E}, 1)$

(h) $(\mathsf{G}(S_A, 0) \land \mathsf{G}(\mathsf{B}, \mathsf{E}, 1)) \to \mathsf{G}(\mathsf{DS})$
$(\mathsf{G}(S_A, 1) \land \mathsf{G}(\mathsf{B}, \mathsf{E}, 0)) \to \mathsf{G}(\mathsf{DS})$

In the syntax of the LWB this is

[4] of course this heavily depends on the machine used; the value given here is just used as a reference for the values that follow.

```
  Alice :=
[
  ~(G(SA_0) & G(SA_1)),

  ~(G(MuA_0) & G(MuA_1)),

  G(A_S_RA_0) -> LG(B_D_RA_0),
  G(A_S_RA_1) -> LG(B_D_RA_1),

  ((G(A_S_SA_pl_MuA_0) <-> G(B_D_SA_pl_MuA_0)) &
     (G(A_S_SA_pl_RA_0) <-> G(B_D_SA_pl_RA_0)),
  (G(A_S_SA_pl_MuA_1) <-> G(B_D_SA_pl_MuA_1)) &
     (G(A_S_SA_pl_RA_1) <-> G(B_D_SA_pl_RA_1)),

  ((G(B_D_SA_pl_RA_0) & G(B_D_RA_0)) v
     (G(B_D_SA_pl_MuA_0) & G(B_D_MuA_0))) -> G(B_D_SA_0),
  ((G(B_D_SA_pl_RA_0) & G(B_D_RA_1)) v
     (G(B_D_SA_pl_MuA_0) & G(B_D_MuA_1))) -> G(B_D_SA_1),
  ((G(B_D_SA_pl_RA_1) & G(B_D_RA_0)) v
     (G(B_D_SA_pl_MuA_1) & G(B_D_MuA_0))) -> G(B_D_SA_1),
  ((G(B_D_SA_pl_RA_1) & G(B_D_RA_1)) v
     (G(B_D_SA_pl_MuA_1) & G(B_D_MuA_1))) -> G(B_D_SA_0),

  (~G(B_E_0) & ~G(B_E_1) & (G(A_E_0) v G(A_E_1))) ->
     (G(B_D_MuA_1) & G(MuA_1)),

  G(B_D_SA_0) -> LG(B_E_0),
  G(B_D_SA_1) -> LG(B_E_1),

  (G(SA_0) & G(B_E_1)) -> G(DS),
  (G(SA_1) & G(B_E_0)) -> G(DS)
];
```

12.3.2 Cheating in Step 3

If Alice wants to cheat in Step 3, she just has to send the wrong value for $S_A \oplus \mu_A$. This alone thus not have any effect, though. Bob cannot deduce anything if he just has this value. For him, it's necessary that his random bit was successfully sent to Alice (i.e. $\mu_A = 1$) and that he somehow knows that. The only way to find that out is, when Alice opens his file before he opens hers. This can be expressed as follows

$$\begin{aligned}(\mathsf{A},\mathsf{C},3) \equiv\ & \neg\mathsf{G}(\mathsf{B},\mathsf{E},0) \wedge \neg\mathsf{G}(\mathsf{B},\mathsf{E},1)\wedge \\ & (\mathsf{G}(\mathsf{A},\mathsf{E},0) \vee \mathsf{G}(\mathsf{A},\mathsf{E},1))\wedge \\ & ((\mathsf{G}(S_A,0) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus \mu_A,0))\vee \\ & (\mathsf{G}(S_A,1) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus \mu_A,1))).\end{aligned}$$

This is written in the LWB as follows

```
    A_C_3 := (~G(B_E_0) & ~G(B_E_1)) &
           (G(A_E_0) v G(A_E_1)) &
           (G(SA_0) & G(A_S_SA_pl_MuA_0)) v
            G(SA_1) & G(A_S_SA_pl_MuA_1)));
```

With the LWB we can now prove, using the theory above

$$(\mathsf{A},\mathsf{C},3) \rightarrow \mathsf{LG}(\mathsf{DS})$$

In the LWB this is

```
    provable(A_C_3 -> LG(DS), Alice);
          true
```

Again, we can prove the same formula with more L-operators on the right hand side, but not with less.

If Alice cheats in step three and she enters the files first, then it is likely that both files are destroyed. Using contraposition this means that—if Alice does not want it to be likely that both files are destroyed—she can not cheat in step three and at the same time use the information she obtained from Bob. Thus, while she can cheat at step three, it's not to here advantage. If she uses the additional information she got, it's likely that both files will be destroyed.

As expected, if we weaken the axiom that Bob will use the password if he knows it, then we get weaker results. We change axiom g) of the theory above to

$$\mathsf{G}(\mathsf{B},\mathsf{D},S_A,0) \rightarrow \mathsf{LLLG}(\mathsf{B},\mathsf{E},0)$$
$$\mathsf{G}(\mathsf{B},\mathsf{D},S_A,1) \rightarrow \mathsf{LLG}(\mathsf{B},\mathsf{E},1)$$

With this changed theory, we can no longer prove $(\mathsf{A},\mathsf{C},3) \rightarrow \mathsf{LG}(\mathsf{DS})$. But we can now prove

$$(\mathsf{A},\mathsf{C},3) \rightarrow \mathsf{LLLG}(\mathsf{DS}).$$

Thus, the less trust Bob gives to the information he deduces from Alice (or the less credibility he assigns to Alice), the less likely it is that the files will be destroyed.

For the LWB, this proof looks as follows

```
    provable(A_C_4 -> LLLLG(DS), WAlice);
          true
```

where `WAlice` is the theory changed as mentioned above. To obtain this result, the LWB computes for quite a long time, using approximately 50 minutes.[5]

[5] without the optimizations of the theory for Alice mentioned above, the same example uses approximately 2 weeks of computation time.

12.3.3 Cheating in Step 2

It is actually not possible for Alice to cheat in step 2, because in that step Alice only sets the value of μ_A. If she does this wrong, then this is completely the same as cheating in step 3.

12.3.4 Cheating in Step 1

Again, if Alice wants to cheat here, she sends a wrong value for S_A, which is the same as cheating in step four (see above).

12.4 Stopping before Step 4

Another way for Alice to cheat would be to stop before step 4, i.e. not sending the last $S_A \oplus R_A$. Surely, Alice cannot stop before step three, or she would not be able to enter Bob's file at all. But, she may already got enough information with step three, i.e. when the oblivious transfer in step one worked. In that case, she could try to withhold the information from step 4. This can be formalized as follows

$$(\mathsf{S},4,0) \equiv \neg\mathsf{G}(\mathsf{B},\mathsf{E},0) \wedge \neg\mathsf{G}(\mathsf{B},\mathsf{E},1) \wedge (\mathsf{G}(\mathsf{A},\mathsf{E},0) \vee \mathsf{G}(\mathsf{A},\mathsf{E},1)) \wedge \mathsf{G}(S_A,0) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus \mu_A,1)$$

for the case that $S_A = 0$ and

$$(\mathsf{S},4,0) \equiv \neg\mathsf{G}(\mathsf{B},\mathsf{E},0) \wedge \neg\mathsf{G}(\mathsf{B},\mathsf{E},1) \wedge (\mathsf{G}(\mathsf{A},\mathsf{E},0) \vee \mathsf{G}(\mathsf{A},\mathsf{E},1)) \wedge \mathsf{G}(S_A,1) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus \mu_A,0)$$

for the case that $S_A = 1$.

Using the LWB syntax this is

```
    S_4_0 := ~G(B_E_0) & ~G(B_E_1) & (G(A_E_0) v G(A_E_1)) &
           G(S_A_0) & G(A_S_SA_pl_MuA_1);

  S_4_1 := ~G(B_E_0) & ~G(B_E_1) & (G(A_E_0) v G(A_E_1)) &
           G(S_A_1) & G(A_S_SA_pl_MuA_0);
```

We can now prove that if Alice really stops before step four and opens Bob's file, then Bob can deduce the password of Alice. This is formalized as

$$(\mathsf{S},4,0) \rightarrow (\mathsf{B},\mathsf{D},S_A,0)$$

and

$$(\mathsf{S},4,1) \rightarrow (\mathsf{B},\mathsf{D},S_A,1).$$

Written in the LWB this is

```
    provable(S_4_0 -> G(B_D_SA_0), Alice);
       true
  provable(S_4_1 -> G(B_D_SA_1), Alice);
       true
```

12.4.1 Bob Cheating

If Bob is trying to cheat, then the situation is completely the same as for Alice. We just need to inverse the roles of Bob and Alice in the whole theory and in the statements to prove.

12.4.2 Obtaining Result

Another thing which is important to know, beside that cheating is not possible, is to make sure that the information that ought to be exchanged really is. There is no use in a protocol which makes it impossible that any of the involved parties can cheat, if the information is not transferred. Thus, we want to show that under the circumstances that if Alice and Bob follow the protocol correctly, then it is at least likely that both can enter the other's files.

First, as before in 12.4, we can show that if Alice enters Bob's file before he enters hers, then Bob can deduce the Password of Alice. On the other hand, if the oblivious transfer from Alice to Bob succeeded, then Bob is even earlier capable of deducing Alice's Password. Thus, if $\mu_B = 1$, then Bob can deduce R_A. Therefore, he should be able to deduce the password of Alice, even if she did not yet enter his file. This can be formalized as follows:

$$\begin{aligned}
&\mathsf{G}(\mathsf{B},\mathsf{D},R_A,0) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus R_A,0) \rightarrow \mathsf{G}(\mathsf{B},\mathsf{D},S_A,0),\\
&\mathsf{G}(\mathsf{B},\mathsf{D},R_A,0) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus R_A,1) \rightarrow \mathsf{G}(\mathsf{B},\mathsf{D},S_A,1),\\
&\mathsf{G}(\mathsf{B},\mathsf{D},R_A,1) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus R_A,0) \rightarrow \mathsf{G}(\mathsf{B},\mathsf{D},S_A,1),\\
&\mathsf{G}(\mathsf{B},\mathsf{D},R_A,1) \wedge \mathsf{G}(\mathsf{A},\mathsf{S},S_A \oplus R_A,1) \rightarrow \mathsf{G}(\mathsf{B},\mathsf{D},S_A,0).
\end{aligned}$$

This is written and executed in the LWB gives us

```
provable(G(B_D_RA_0) & G(A_S_SA_pl_RA_0) -> G(B_D_SA_0), Alice);
     true
provable(G(B_D_RA_0) & G(A_S_SA_pl_RA_1) -> G(B_D_SA_1), Alice);
     true
provable(G(B_D_RA_1) & G(A_S_SA_pl_RA_0) -> G(B_D_SA_1), Alice);
     true
provable(G(B_D_RA_1) & G(A_S_SA_pl_RA_1) -> G(B_D_SA_0), Alice);
     true
```

Thus, all in all we have shown that if there is no cheating, Bob and Alice will likely be able to deduce the other's passwords. Furthermore, we have also shown that if either of both cheats, then its likely that both files will be destroyed.

12.4.3 Analysis of the the Axioms

In their paper [22], Rabin and Halpern take a closer look at some of the axioms used above. We won't repeat all the theory here, but just point out how the these things can be proven using the Logics Workbench.

Axiom (e)

This axiom says that Bob can deduce $\mu_A = 1$ if Alice enters his file before he enters hers. This deduction process can be captured with the following axioms:

$$\begin{array}{lll}
(8A) & \mathsf{G}(\mathsf{A},\mathsf{E},0) \rightarrow \mathsf{G}(\mathsf{A},\mathsf{D},S_B,0) & \\
 & \mathsf{G}(\mathsf{A},\mathsf{E},1) \rightarrow \mathsf{G}(\mathsf{A},\mathsf{D},S_B,1) & \\
(9A) & \mathsf{G}(\mathsf{A},\mathsf{D},S_B,0) \quad \rightarrow & [(\mathsf{G}(\mathsf{A},\mathsf{D},S_B \oplus R_B,0) \wedge \mathsf{G}(\mathsf{A},\mathsf{D},R_B,0)) \quad \vee \\
 & & (\mathsf{G}(\mathsf{A},\mathsf{D},S_B \oplus R_B,1) \wedge \mathsf{G}(\mathsf{A},\mathsf{D},R_B,1)) \quad \vee \\
 & & (\mathsf{G}(\mathsf{A},\mathsf{D},S_B \oplus \mu_B,0) \wedge \mathsf{G}(\mathsf{A},\mathsf{D},\mu_B,0)) \quad \vee \\
 & & (\mathsf{G}(\mathsf{A},\mathsf{D},S_B \oplus \mu_B,1) \wedge \mathsf{G}(\mathsf{A},\mathsf{D},\mu_B,1))], \\
 & \mathsf{G}(\mathsf{A},\mathsf{D},S_B,1) \quad \rightarrow & [(\mathsf{G}(\mathsf{A},\mathsf{D},S_B \oplus R_B,1) \wedge \mathsf{G}(\mathsf{A},\mathsf{D},R_B,0)) \quad \vee \\
 & & (\mathsf{G}(\mathsf{A},\mathsf{D},S_B \oplus R_B,0) \wedge \mathsf{G}(\mathsf{A},\mathsf{D},R_B,1)) \quad \vee \\
 & & (\mathsf{G}(\mathsf{A},\mathsf{D},S_B \oplus \mu_B,1) \wedge \mathsf{G}(\mathsf{A},\mathsf{D},\mu_B,0)) \quad \vee \\
 & & (\mathsf{G}(\mathsf{A},\mathsf{D},S_B \oplus \mu_B,0) \wedge \mathsf{G}(\mathsf{A},\mathsf{D},\mu_B,1))] \\
(10A) & \neg\mathsf{G}(\mathsf{A},\mathsf{D},\mu_B,0) \wedge (\neg\mathsf{G}(\mathsf{B},\mathsf{E},0) \wedge \neg\mathsf{G}(\mathsf{B},\mathsf{E},1) \rightarrow \neg\mathsf{G}(\mathsf{A},\mathsf{D},\mu_B,1)) &
\end{array}$$

Axiom (8A) says that Alice is rational, i.e. Alice only enters Bob's file if she can deduce his password. Axiom (9A) gives the modular arithmetic used for deducting the value of S_B. Then, axiom (10A) describes under which circumstances Alice can deduce the value of μ_B.

Using axioms (8A), (9A) and (10A) we can show that Alice can deduce R_B, if Bob enters her file before she his, i.e.

$$(\mathsf{A},\mathsf{E},\mathsf{F}) \rightarrow \mathsf{G}(\mathsf{A},\mathsf{D},R_B,0) \vee \mathsf{G}(\mathsf{A},\mathsf{D},R_B,1).$$

In the LWB this proof looks as follows:

```
Ax8A  := (G(A_E_0) -> G(A_D_SB_0)) &
           (G(A_E_1) -> G(A_D_SB_1));

  Ax9A  := (G(A_D_SB_0) -> ((G(A_D_SB_pl_RB_0) & G(A_D_RB_0))   v
                            (G(A_D_SB_pl_RB_1) & G(A_D_RB_1))   v
                            (G(A_D_SB_pl_MuB_0) & G(A_D_MuB_0)) v
                            (G(A_D_SB_pl_MuB_1) & G(A_D_MuB_1)))) &
           (G(A_D_SB_1) -> ((G(A_D_SB_pl_RB_1) & G(A_D_RB_0))   v
                            (G(A_D_SB_pl_RB_0) & G(A_D_RB_1))   v
                            (G(A_D_SB_pl_MuB_1) & G(A_D_MuB_0)) v
                            (G(A_D_SB_pl_MuB_0) & G(A_D_MuB_1))));

  Ax10A := ~G(A_D_MuB_0) & (~G(B_E_0) & ~G(B_E_1) -> ~G(A_D_MuB_1));

  A_E_F := ~G(B_E_0) & ~G(B_E_1) & (G(A_E_0) v G(A_E_1));

  provable((A_E_F) -> (G(A_D_RB_0) v G(A_D_RB_1)),
           [ Ax8A, Ax9A, Ax10A ]);
       true
```

We leave out further considerations of the paper dealing with knowledge operators because these operators are not implemented in the Logics Workbench (we'd have to make a mix between the logic of likelihood and S_4).

Chapter 13

Implementation

This chapter examines the implementation of the automatic theorem prover in LL^- that was done for the Logics Workbench. It implements the double-sided sequent calculus with loop-check mentioned in chapter 11

13.1 Introduction

Aside from the code that needed be written in order to have a prover for LL^-, the structure of the LWB made other coding necessary as well. First, because the logic of likelihood needs the additional modal operators `L`, `K`, `F`, `G`, the parser of the LWB had to be enhanced. To be able to enter formulas with these new operators, the parser has to know these operators and does also need to know how to deal with them.

The LWB makes things a bit more complicated, because the user can use different logics at the same time. Because the new operators are normal letters, it would limit users too much if we treat them as modal operators in all logics. This would prohibit the user from using symbol names like `Group`, or `Logic`, for example. These words use letters that are also symbols for operators and because the LWB does not require a space after an operator, they would be interpreted as `G roup` and `L ogic`, respectively[1]. Thus, we had to limit the use of these operators to the logic of likelihood itself, as it was also done with temporal operators. These operators are only valid when the module `llh` is the current top module.

Furthermore, the parser of the LWB was changed to allow these new, letter style operators only at the beginning of a word. Thus, even if a letter is an operator of a logic, it may still appear in the middle of a symbol name (e.g. `STL` or `PROOF`). Previously, symbols like these produced a general parsing error, because the letters were interpreted as operators.

[1] requiring no spaces, on the other hand, allows to directly write formulas like `LLGF A`, instead of having to write `L L G F A`.

13.1.1 LWB Functions

In the LWB, the implementation of a prover for a new logic also requires the creation of a new module. The module only requires some initialization code, but all the modules already present in the LWB provide a common set of basic functions. Of course, a new logic has to implement these functions as well.[2] The following functions were implemented for LL^-:

`arrange`	Arranges the sub formula in a formula or theory by using associativity and commutativity.
`convert`	Converts connectives from one type to another (for example replacing all implications by disjunctions).
`depth`	Computes the length of the longest branch of a formula viewed as a tree.
`length`	Computes the length of the formula (as defined in a previous chapter).
`less`	Determine if a formula is lexicographically less than another one (allows sorting of formulas).
`nnf`	Compute the negation normal form of a formula given.
`nnfp`	This function computes a negation normal formula of the given formula, using new variables to make the result much shorter.
`provable`	This is the prover, which we be detailed below.
`randomfml`	Compute a random formula for LL^-.
`remove`	Remove as much constants (true and false) from the formula as possible.
`sort`	Sorts a set of formula lexicographically.
`subfmls`	Determines all sub formulas (as defined in an earlier chapter) of the given formula
`typek`	Test the type of the given expression.
`vars`	Extract all the names of variables from the given formula.

Below, we will concentrate on the `provable` function only. The implementations of the other functions is similar to those of other logics.

13.2 Overview

The implementation for the prover for the logic of likelihood differs quite a bit from the implementations that were done for other logics in the LWB. While the theoretical side is comparable to other logics, especially as shown earlier to the modal logics S_4 and KT, the practical side is quite different. The implementation for this prover is done using object oriented principles and makes heavy use of the standard template library (STL [44]). This makes the code much easier to read and also allows easier extension of the code.

[2] the structure of the LWB does not allow sharing of such functions between different modules, i.e. different logics. While this would be possible using the same techniques mentioned below for the prover, it would require some big redesign of the LWB.

13.2.1 Object Oriented Programming

We won't take a closer look on how object oriented programming works or how programming in C++ is done (c.f. [34], [19]). In the following we assume that the reader is somewhat familiar with object oriented programming in C++.

13.3 Classes

This section will take a look at the objects that were defined for the prover. Because the standard template library was used, it was not necessary to implement many objects. The sets and lists that are offered by the STL are sufficiently efficient[3] and powerful.

13.3.1 Main Class

There is only one main class, namely `llh_ProofNode`.[4] This class represents a node in the proof tree of a formula. It contains all data necessary to represent a sequent of the sequent calculus. It provides methods to add formulas to the sequent and to check if a sequent is provable. To check this, the node can create additional proof nodes, for example when a branching rule is encountered.

Internally, the class provides methods for each rule of the calculus, although some of these rules are hardly recognizable. It uses these rules to check the current node for provability and to add additional formulas. The class furthermore has methods to support branching rules, backtracking, use check, the history and printing of algorithmic information. These parts of the proof node will be detailed below.

Each proof node stores all the formulas of a sequent of the calculus as mentioned in chapter 11. While the marked formulas are stored in simple sets, the formulas to the left and to the right of the sequent delimiter are split up in different groups. A first group contains all formulas consisting of variables. A second group consists of those formulas that require branching rules, like the $\wedge$-formulas on the right side of the sequent. A last group contains all formulas requiring backtracking, for example the G-formulas on the right side. All other formulas are split up (cf. 13.4) using `classify()` until all their parts end up in one of those groups.

This split into different groups allows more efficient treatment of the formulas involved, because at the different steps of the algorithm different types of formulas are used.

[3] this heavily depends on the actual implementation provided by the compiler; the implementation of the GNU C++ compiler is about 10 times faster than the one present in the Sun CC compiler—at least for those version that were available at that time.

[4] in the LWB, all the names of a module are prefixed with the module name to prevent name space cluttering; nowadays, this could be achieved easier by using name spaces or by using a class for each module.

13.3.2 Auxiliary Class

`llh_Expression` is the only auxiliary class (not counting the STL classes), used by the prover. This class is actually just a wrapper for the internal `expr` class provided by the LWB kernel. Because the LWB kernel and its expression class are not compatible to the STL classes, this wrapper was necessary.[5]

13.3.3 Nested Classes

The `llh_ProofNode` class uses some internal, nested classes. They are presented here for a better understanding of the main class, but are actually quite simple.

`ProofRule`

This class stores a pointer to a method for a specific rule of the calculus. While a simple method pointer might work as well, this class additionally takes care of empty or undefined method pointers. Otherwise, the class is made in a way that it can be used exactly like a method itself.

`History`

This nested class is essential for the loop check. The history class stores a list of states encountered during proof search. These lists are then stored, together with their main formula, in the main class `llh_ProofNode`. It exactly represents the histories H_K and H_L of the double-sided sequent calculus.

13.4 Main Algorithm `provable()`

Because some auxiliary classes and many methods are available, the main proving algorithm of the `llh_ProofNode` class is actually quite simple. A description of the rules and methods used in the main algorithm follows in later sections. To make the algorithm easier to read, use check, infolevel output and percents statements are left out. Furthermore, all code dealing with histories is ignored for now as well. All these parts of the final algorithm will be detailed in later sections.

The main algorithm can be written in pseudo code as follows:

1. Add the formula to prove to the node, if necessary including the theory to use, by using `classify()`.[6]

[5] it would be much work to replace the internal `expr` class with a STL compatible, modern class, because all the algorithms would have to be changed as well.

[6] this step is actually done outside the `llh_ProofNode` class.

```
bool
llh_ProofSequence::provable()
{
  static Position lastSide    = Expression::right;

  lastSide                    = other(lastSide);
  set<Expression> &firstSplit = split(lastSide);

  if(!firstSplit.empty())
  {
    Expression formula = *(firstSplit.begin());

    firstSplit.erase(firstSplit.begin());

    return split(formula, lastSide);
  }

  lastSide                     = other(lastSide);
  set<Expression> &secondSplit = split(lastSide);

  if(!secondSplit.empty())
  {
    Expression formula = *(secondSplit.begin());

    secondSplit.erase(secondSplit.begin());

    return split(formula, lastSide);
  }

  // now its time to try the backtracking stuff
  while(!lBacktrack.empty())
  {
    Expression formula = *(lBacktrack.begin());

    lBacktrack.erase(lBacktrack.begin());

    // save the old state for backtrack
    llh_ProofSequence old(*this);

    if(backtrack(formula, Expression::left))
      return true;

    // restore the old state
    *this = old;
  }

  while(!rBacktrack.empty())
  {
    Expression formula = *(rBacktrack.begin());

    rBacktrack.erase(rBacktrack.begin());

    // save the old state for backtrack
    llh_ProofSequence old(*this);

    if(backtrack(formula, Expression::right))
      return true;

    // restore the old state
    *this = old;
  }

  return false;
}
```

Table 13.1: Main proofing algorithm

```
bool
llh_ProofSequence::classify(const Expression &inFormula,
                            Position inPos)
{
  if(present(inFormula, other(inPos)))
    return true;

  return classifyFuncs[inFormula.type()](*this, inFormula, inPos);
}
```

Table 13.2: Main classification function

2. If there is a branching formula, call `split()` for it and return the result.

3. If there are no splitting formulas, then using `backtrack()` check in turn each backtracking formula, restoring the state after each formula.

4. Return true if a backtracking formula is found which is provable, return false else.

The methods used in this algorithm will be detailed below. Each of these methods calls a rule of a specific category to actually carry through the rule of the calculus. The actual implementation of this main algorithm, excluding step 1, is shown in table 13.1.

Classification with `classify()`

Classification is automatically done each time a formula is added to the current sequent. Thus, no formulas are directly added to some internal storage, but instead all new formulas are classified using the `classify()` method. This method makes sure each formula is split up and distributed to their appropriate storage spaces, i.e. either variables, branching formulas or backtracking formulas.

Again, use check, infolevel, percents, and some special cases (the constants true and false) have been left out). The classification method as it is implemented is shown in table 13.2. As can be seen, it first checks if the formula is present[7] in the other side of the sequent and then calls the function associated with the main operator of the given formula. This is done using a previously prepared calling table to speed up rule selection and invocation.

```
bool
llh_ProofSequence::split(const Expression &inFormula, Position inPos)
{
  // first we check for axioms (this was already done when inserting
  // the formula into the list, but meanwhile other formulas may have
  // been added and if we have an axiom we can leave out quite a bit)
  if(present(inFormula, other(inPos)))
    return true;

  return splittingFuncs[inFormula.type()](*this, inFormula, inPos);
}
```

Table 13.3: Main split function

```
bool
llh_ProofSequence::backtrack(const Expression &inFormula,
                             Position inPos)
{
  return backtrackingFuncs[inFormula.type()](*this, inFormula, inPos);
}
```

Table 13.4: Main backtrack function

Splitting with `split()`

The `split()` method is similar to `classify()`. It just needs to check again for axioms—just in case a new formula was inserted without the necessary check—and then calls the splitting function associated with the top-level operator of the formula.

The simplified code for this method is shown in table 13.3.

Backtracking with `backtrack()`

The main backtracking method `backtrack()` is depicted in table 13.4. It is a very simple method, consisting only of a call to the operator dependent backtracking function. In this case, a check for an axiom is not necessary.

[7] the `present()` method does a bit more than just check if the whole formula appears on the other side; if the given formula is one that would have been split into sub formulas when classified on the other side, then the function checks for existence of these sub formulas instead; while that way the function is a bit more complicated and uses a bit more time, the overall speed gain is considerable.

13.5 Rules

The rules of the sequent calculus are represented in the implementation by various functions, one method for each rule. Such a function takes the current state of the object (i.e. the current node in the proof tree) and modifies it to a new state, if necessary by creating additional nodes if a rule is a branching rule. Thus, a rule method only has to do a single step of the proof, just as in the calculus. This makes writing, and if necessary, enhancing a rule quite easy.

Because a rule is a normal C++ method, it is also not limited in what it can do. If the rules were written more abstractly using a special syntax instead of a real method, they would be limited. As a method, there are no limits and if a rule needs to do additional things, like using heuristics or other optimizations, then this is possible as well while processing the rule.

As mentioned in the description of the main algorithm, the rules are split into three categories. This distinction of the rules is necessary to be able to decide in the main algorithm which rule has to be called at which time. For each category, there is method for each top-level operator a formula treated with such a rule might have.

13.5.1 Classification Rules

Classification rules are the simplest of the rules. They represent invertible rules of the calculus where no branching or backtracking occurs. They do nothing else than putting a given formula into its storage place or, if necessary, splitting the formula and recursively call another classification rule.

These rules are always called when a new formula has to be stored. The first time this is the case when the proofing process is started and the formula to be proven has to be stored. Then, each time one of the rules creates a new formula, for example because a formula is split up, another classification rule is called. Each classification rule returns true if an axiom is found and false else.

Because the prover must be able to store and thus classify every possible formula, there is a classification method for each operator.

Example 17 ***Classification of Disjunctions***
As an example, we take a closer look of the classification function for the disjunction.[8] This function is called for all formulas whose top operator is a disjunction. The code is shown in table 13.5.

Like all classification methods, it takes as argument the expression, i.e. formula, that is to be classified, and the position this formula occurs, i.e. either on the left or the right side in the sequent. The function returns true, if the classification results in an axiom, i.e. a formula appears on both sides of the sequent. In all other cases the function returns false.

[8] some error checking has been removed from the code to make it clearer.

```
inline bool
llh_ProofSequence::classifyOr(const Expression &inFormula,
                              Position inPos)
{
  if(inPos == Expression::left)
  {
    lSplit.insert(inFormula);

    return false;
  }
  else
    return classify(inFormula[Expression::left], Expression::right) ||
           classify(inFormula[Expression::right], Expression::right);
}
```

Table 13.5: Implementation of the classification rule for the disjunction

We have to distinguish two cases, depending on which side the formula has to be stored. If the formula is on the left side, then we'd actually have to call the (l-$\vee$)-Rule, which is a branching rule. Because in the classification step we don't want to carry through any branching rules, we just store the formula in a set for later treatment with a splitting rule (see below). If the formula is added on the right side, we carry through the full (r-$\vee$)-rule, because the rule is invertible and not branching. In that case, according to the rule, we just need to store both operands of the top level disjunction. These subformulas are classified again. We have an axiom, if either classifying the left or right operand produced an axiom.

13.5.2 Splitting Rules

While classification rules can always be done first, because they don't have a long execution time, splitting rules are worse in that aspect. Splitting rules and their functions implement the branching rules of the sequent calculus (i.e. mainly (l-$\vee$) and (r-$\wedge$) rules). These rules always start a new branch for proving and thus require more computation time. Therefore, splitting rules are done only after no more classification rules can be applied, but before starting with backtracking rules.

Splitting rules have to be defined for all rules of the sequent calculus which have more than one premiss. This means that there has to be a splitting function for all operators that can appear as top level operator of the main formula of such a rule. This mainly excludes the modal operators and the negation, i.e. all unary operators.

Example 18 ***Splitting of Conjuctions***
As an example, we take a look at the splitting function for the conjunction[9], shown in table 13.6.

[9] all pieces of code that deal with status output and the computation of the amount of percents already done has

```
bool
llh_ProofSequence::splitAnd(const Expression &inFormula,
                            Position inPos)
{
  llh_ProofSequence sideBranch(*this);

  return provable(inFormula[Expression::left], Expression::right))
         &&
         sideBranch.provable(inFormula[Expression::right],
                             Expression::right);
}
```

Table 13.6: Implementation of the splitting rule for the conjunction

This function just copies the current node and tries to proof each subformula of the premiss in a different, independent node. The result is true if both calls return true and false else.

13.5.3 Backtracking Rules

The backtracking rules are used for all rules requiring backtracking, i.e. the G/K-rules. The real backtracking is done in the main algorithm. These methods just have to adjust the sequent as determined in the rule of the calculus.

Example 19 ***Backtracking of*** **L*-formulas***

As an example, we take a quick look at the rule for the L-operator, show in table 13.7. As with the previous examples, some additional code—like use check and infolevel output—has been left out. This function looks quite complicated, but in fact its just some moving of sets using the STL routines.[10]. In fact, this method just clears the appropriate sets and transfers some formulas from the marked to the unmarked state using `classify()`.

13.6 History

Section 11.8.2 shows that it is necessary to add a history to the rules to be able to check for loops and thus obtain a terminating algorithm. For clarity, the history has been left out in the example code above, but of course has to be present.

been removed from this demonstration code, as well as some error checking and the use check; the real code is about 10 times the size of the one shown.

[10] we cannot use some of the generic algorithms `for_each` or `set_union` here, because if the classification finds an axiom we can stop; the STL algorithms would do all elements in any case and thus would be less efficient; furthermore, it's not possible to call member functions in a `for_each` statement.

```
bool
llh_ProofSequence::backtrackL(const Expression &inFormula,
                              Position inPos)
{
  // remove the whole left and right sequents
  lMisc.clear();
  rMisc.clear();
  lSplit.clear();
  rSplit.clear();
  lBacktrack.clear();
  rBacktrack.clear();

  // add the whole lL, we use a copy of the list here,
  // otherwise things added may be removed again in the
  // loop below.
  set<Expression> lLOld;
  lLOld.swap(lL);

  while(!lLOld.empty())
  {
    Expression formula = *(lLOld.begin());

    lLOld.erase(lLOld.begin());

    if(classify(formula, Expression::left))
      return true;
  }

  // add the whole rL, as above
  set<Expression> rLOld;
  rLOld.swap(rL);

  while(!rLOld.empty())
  {
    Expression formula = *(rLOld.begin());

    rLOld.erase(rLOld.begin());

    if(classify(formula, Expression::right))
      return true;
  }

  // add the whole lF
  for(set<Expression>::iterator i = lF.begin(); i != lF.end(); i++)
  {
    if(classify(*i, Expression::left))
      return true;
  }

  // add the whole rF
  for(set<Expression>::iterator i = rF.begin(); i != rF.end(); i++)
  {
    if(classify(*i, Expression::right))
      return true;
  }

  return provable(inFormula[Expression::left], Expression::left);
}
```

Table 13.7: Implementation of a backtracking rule

As can be seen from the rules of the sequent calculus, there are four different sets of histories necessary, namely H_L, H_F, H_K, and H_G. Basically, the history contains the complete state of a node, to be able to recognize it later. The history only depends on the premiss of the rule, and thus, because it is only used in G/K-rules, only the marked formulas are necessary to identify it.

The histories for the G and F formulas only need to store the formulas encountered, because these histories can only increase and are cleared when a (r-F) or (l-G)-rule is used. They are implemented using STL sets.

The histories for the K and L-rules are more complicated. Because these histories can change arbitrarily, it is necessary to store the sets of marked formulas, i.e. the complete premiss of the rule, along with the main formula of the rule. Only if the same sets and main formula are encountered again, we have detected a loop. These histories are stored using the class.

13.6.1 Code

The following additions to the code have to be made in order to include history handling:

- the classification functions for F and G have to clear all the history sets when a new formula is classified,
- before starting backtracking in `provable()`, all backtracking formulas are inserted into the history[11],
- all the rules of the calculus with history checks have to make them before actually applying the rule

These changes are all that is necessary to add loop check to the code. Actually, according to the rules of the calculus, it would have been necessary to adjust the histories in each backtracking rule. But because we add all backtracking formulas before we start backtracking, we only need to do that once, which is much easier to implement and more efficient as well.

13.7 Use Check

Use check was also mentioned in [29] and originally comes from [41], where it was used for a decision procedure for intuitionistic logic. We take a look here to see what changes have to be incorporated in the code to implement use check for LL^-. Contrary to other implementations in the LWB, use check for LL^- is done using object oriented techniques. Because the whole algorithm uses object oriented programming, it is easy to add used check into an already working algorithm.

Use check is mainly done in two steps. In a first step, every axiom found adds its main formulas to sets of used formulas. For simple propositional variables this means just adding the formula

[11] this optimization increases computation speed much.

for more complicated formulas this means adding all relevant sub formulas to their appropriate sets as well. This way, all formulas that were used to actually obtain the axiom are stored.

After a successful branch of a splitting rule, a special function checks if the main formula of the branch has been used in the proof. If this is not the case, then the second branch of the rule does not have to be proven at all, saving much computation time.

If use check is not used, then the example in 12.3.1, checking for cheating in step 4, requires more than 80 hours to compute the result, instead of the approximately 30 seconds it takes with use check.

13.8 Infolevel

The Logics Workbench has a special way for printing information about a running algorithm, the so called infolevel output. This output can be used to see what internally happens when an algorithm runs, especially what a prover does when trying to prove a formula. The generated output represents how the the algorithm internally develops, not specially prepared information showing a readable proof. Nevertheless, the infolevel output can be interesting to find out why something is provable or why it is not. Furthermore, when dealing with errors in the prover or when debugging the prover, the infolevel is a great help for finding bugs.

Adding infolevel output to the algorithm just requires some additional outputs statements. By using a special method, this printing also be reused for future provers.

Example 20 ***Infolevel***
As an example of the infolevel output we look at the output generated when proving the formula $\mathsf{G}(A \wedge B) \leftrightarrow (\mathsf{G}A \wedge \mathsf{G}B)$. In the LWB this is written as follows

```
set("infolevel", 5);
  provable(G(A & B) <-> GA & GB);
```

The first line sets the infolevel output to level 5, determining that all information of the main algorithm is to be printed. The output generated is shown in table 13.8. It starts with the name of the rule used, using the following abbreviations:

`C`	for a classification rule,
`S`	for a splitting rule,
`B`	for a backtracking rule.

This is followed by `'l'` if the rule is applied on the left or by `'r'` if the rule is applied on the right side of the sequent. The name of the rules is ended by the top level operator of the formula, thus completely identifying the method to be called for the rule.

After the identification of the rule follows the sequent as it is before applying the rule. The main formula of the rule is enclosed in quotes. The sequents are written as in the theory chapter, i.e.

the marked formulas at the beginning and the end. Finally follows a list of those formulas that were used for proving so far. This information is used for the use check.

The output is indented to show the two branches involved in a splitting rule and to make clear that backtracking rules are actually on the same level and are just carried through one after the other.

For technical reasons, the infolevel output is printed before the rule is actually carried through. This means that the information given for the used formulas does not incorporate the information from the rule printed[12]

13.9 Derivation

The prover is implemented in a way allowing its derivation. Thus, the same structure can be used for provers for other logics. As long as the general proving principle for a logic is the same, the prover can be derived and most of the implementation can be reused.

Of course, for some logics—namely classical propositional logic—it would be better to create a new prover from the current one and make the prover for LL^- a derivation of it. Because classical logic does not need loop checks and backtracking, such a prover is simpler and thus could be more efficient. It should then be no problem to create a derivation of this prover to get a prover for the logic of likelihood.

For such a derivation, mainly the new rule functions (as far as they differ) and history and backtracking have to be added. But things like use check or infolevel printing and all the necessary interfacing to the LWB can be taken over without change. Furthermore, by incorporating additional optimizations into the base prover, they can be done for all provers at once, .

13.10 Efficiency

An important question when using object oriented techniques is always their efficiency. It is widely believed that code written using object oriented methods has to be much slower and inefficient than code written without. This does not always have to be the case (cf. [33]).

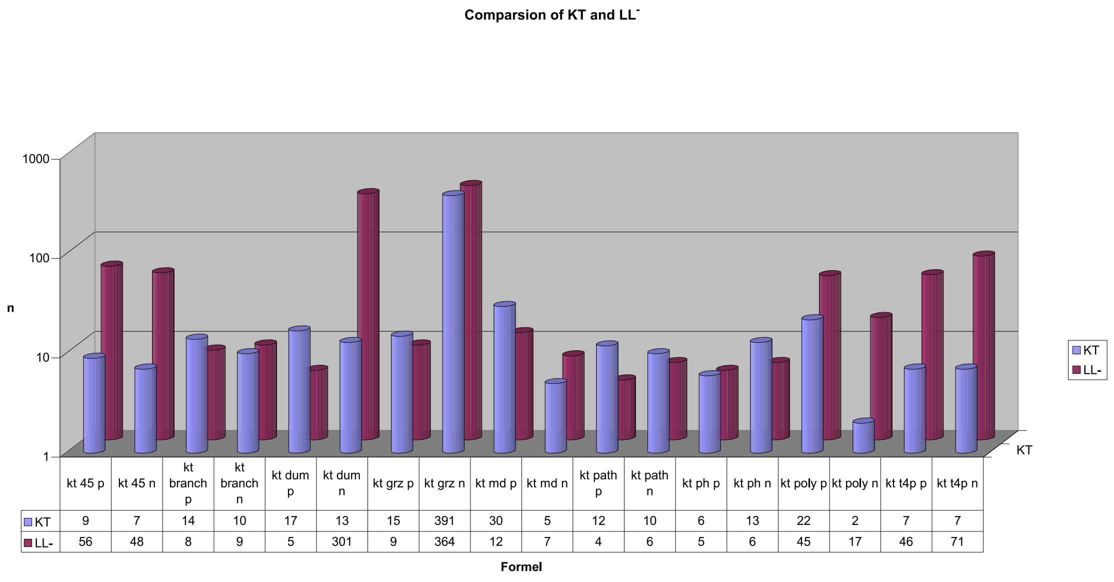
As shown in 11.9, S_4 and KT can easily be embedded into LL^-. As a test for efficiency, we use the benchmark formulas of [2] for KT and S_4 with an appropriate translation into LL^-. This results are compared to the results for existing provers for S_4 and KT already present in the LWB.

Figures 13.1 and 13.2 show the maximal type of formula that could be proven in less than 100 seconds. As can be seen, the prover for LL^- is generally somewhat slower than the existing

[12] thus the formulas used in the axioms don't show up in the printing of the used formulas. Furthermore, in the given example, no used formulas are shown at all, because two splitting rules follow each other, and each branch of a splitting rule uses a new set of used formulas.

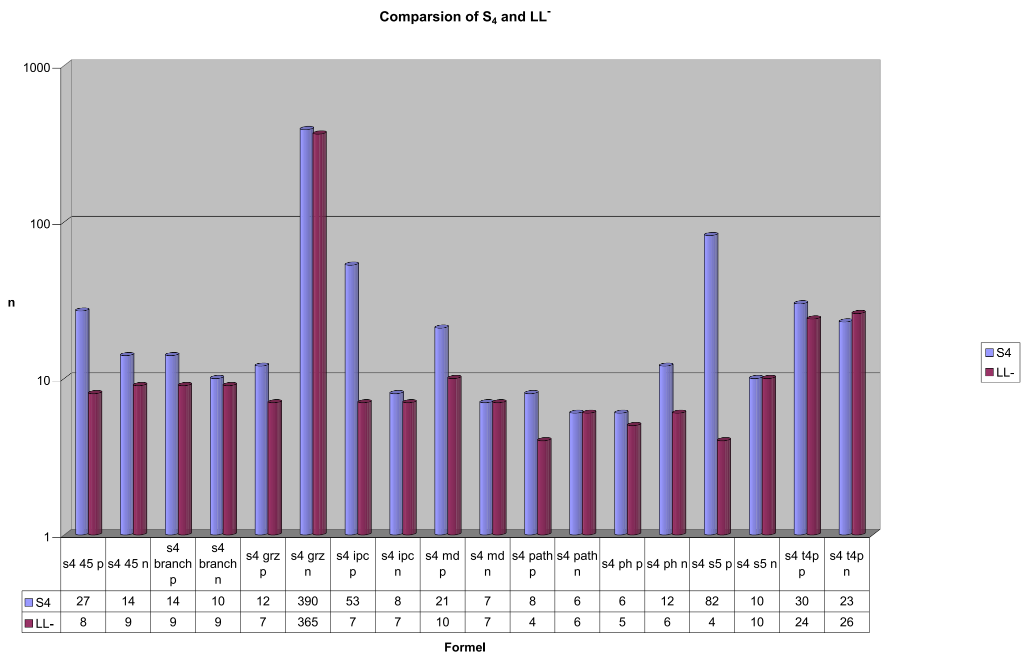
```
Cr-<-> ;  | ; ;  =) "G (A & B) <-> G A & G B"; ;  | ;   [ =) ]
  Sr-<-> ;  | ; ;  =) ; "G (A & B) <-> G A & G B";  | ;   [ =) ]
    Cl-G   ;  | "G (A & B)"; ;  =) ; ;  | ;   [ =) ]
    Deleting History for G (A & B)
    Cl-&   ; A & B | "A & B"; ;  =) ; ;  | ;   [ =) ]
    Cl-var ; A & B | "A"; ;  =) ; ;  | ;   [ =) ]
    Cl-var ; A & B | "B", A; ;  =) ; ;  | ;   [ =) ]
    Cr-&   ; A & B | A, B; ;  =) "G A & G B"; ;  | ;   [ =) ]
    Sr-&   ; A & B | A, B; ;  =) ; "G A & G B";  | ;   [ =) ]
      Cr-G   ; A & B | A, B; ;  =) "G A"; ;  | ;   [ =) ]
      Br-G   ; A & B | A, B; ;  =) ; ; "G A" | ;   [ =) ]
        Cl-&   ; A & B | "A & B"; ;  =) ; ;  | ;   [ =) ]
        Cl-var ; A & B | "A"; ;  =) ; ;  | ;   [ =) ]
        Cl-var ; A & B | "B", A; ;  =) ; ;  | ;   [ =) ]
        Cr-var ; A & B | A, B; ;  =) "A"; ;  | ;   [ =) ] (axiom)
        (axiom)
      ------------------------------
      Cr-G   ; A & B | A, B; ;  =) "G B"; ;  | ;   [ =) ]
      Br-G   ; A & B | A, B; ;  =) ; ; "G B" | ;   [ =) ]
        Cl-&   ; A & B | "A & B"; ;  =) ; ;  | ;   [ =) ]
        Cl-var ; A & B | "A"; ;  =) ; ;  | ;   [ =) ]
        Cl-var ; A & B | "B", A; ;  =) ; ;  | ;   [ =) ]
        Cr-var ; A & B | A, B; ;  =) "B"; ;  | ;   [ =) ] (axiom)
        (axiom)
      (axiom)
    ------------------------------
    Cr-G   ;  | ; ;  =) "G (A & B)"; ;  | ;   [ =) ]
    Cl-&   ;  | "G A & G B"; ;  =) ; ; G (A & B) | ;   [ =) ]
    Cl-G   ;  | "G A"; ;  =) ; ; G (A & B) | ;   [ =) ]
    Deleting History for G A
    Cl-var ; A | "A"; ;  =) ; ; G (A & B) | ;   [ =) ]
    Cl-G   ; A | "G B", A; ;  =) ; ; G (A & B) | ;   [ =) ]
    Deleting History for G B
    Cl-var ; A, B | "B", A; ;  =) ; ; G (A & B) | ;   [ =) ]
    Br-G   ; A, B | A, B; ;  =) ; ; "G (A & B)" | ;   [ =) ]
      Cl-var ; A, B | "A"; ;  =) ; ;  | ;   [ =) ]
      Cl-var ; A, B | "B", A; ;  =) ; ;  | ;   [ =) ]
      Cr-&   ; A, B | A, B; ;  =) "A & B"; ;  | ;   [ =) ] (axiom)
      (axiom)
    (axiom)
          true
```

Table 13.8: Infolevel output for the proof of $\mathsf{G}(A \wedge B) \leftrightarrow (\mathsf{G}A \wedge \mathsf{G}B)$



	kt 45 p	kt 45 n	kt branch p	kt branch n	kt dum p	kt dum n	kt grz p	kt grz n	kt md p	kt md n	kt path p	kt path n	kt ph p	kt ph n	kt poly p	kt poly n	kt t4p p	kt t4p n
KT	9	7	14	10	17	13	15	391	30	5	12	10	6	13	22	2	7	7
LL-	56	48	8	9	5	301	9	364	12	7	4	6	5	6	45	17	46	71

Figure 13.1: Comparison of benchmark formulas between KT and LL$^-$



	s4 45 p	s4 45 n	s4 branch p	s4 branch n	s4 grz p	s4 grz n	s4 ipc p	s4 ipc n	s4 md p	s4 md n	s4 path p	s4 path n	s4 ph p	s4 ph n	s4 s5 p	s4 s5 n	s4 t4p p	s4 t4p n
S4	27	14	14	10	12	390	53	8	21	7	8	6	6	12	82	10	30	23
LL-	8	9	9	9	7	365	7	7	10	7	4	6	5	6	4	10	24	26

Figure 13.2: Comparison of benchmark formulas between S_4 and LL$^-$

Formula	**CPC**	**LLH**	**CPC sorted**	**LLH sorted**
`pigeonhole(2)`	0.01 s	0.00 s	0.00	0.00
`pigeonhole(3)`	0.00 s	0.01 s	0.01 s	0.01 s
`pigeonhole(4)`	0.04 s	0.19 s	0.18 s	0.30 s
`pigeonhole(5)`	0.72 s	3.72 s	1.97 s	1.35 s
`pigeonhole(6)`	15.07 s	1 m 26.23 s	6 m 54.37 s	5 m 40.20 s

Table 13.9: Comparison of pigeonhole formulas

algorithms, but not in all cases. Further analysis, using classical formulas help to find out where the time is lost. Figure 13.9 shows the execution time for some pigeonhole formulas. While the classical prover is always much faster than the one for LL^-, this changes when the formula is specially sorted. In that case, the classical prover uses much more time. While the prover for LL^- also requires more time, it is now faster than the classical one. The reason for this behavior lies in the fact, that the implementation for LL^- uses STL sets to store formulas. These sets internally sort their elements, thus sorting is always done and thus not dependent on the sequence of the input formula. This sorting uses quite a lot of computation time, but can sometimes improve the speed as well.

The object oriented prover could be much improved if the sorting used for the sets could either be improved in speed or could be heuristically changed to always prove formulas with the most chance of success first. Unfortunately, this cannot be implemented by adjusting the STL provided `set` class. Instead, a complete new class for sets must be implemented.

Appendix A

Bibliography

[1] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1998.

[2] Peter Balsiger, Alain Heuerding, and Stefan Schwendimann. A benchmark method for the propositional modal logics K, KT, S4. *Journal of Automated Reasoning*, 24(3):297–317, 2000.

[3] Charles H. Bennet, Gille Brassard, Claude Crépau, and Marie-Hélène Skubiszewska. Practical quantum oblivious transfer, 1992.

[4] Peppo Brambilla. Sequent calculus proof representation. Master's thesis, University of Bern, 1996.

[5] Apple Computer. Inside Macintosh. `http://developer.apple.com/techpubs/mac/mac.html`.

[6] Apple Computer. *Inside Macintosh: Overview*. Addison-Wesley, 1992.

[7] Apple Computer. *Inside Macintosh: Devices*. Addison-Wesley, 1993.

[8] Apple Computer. *Inside Macintosh: Interapplication Communication*. Addison-Wesley, 1993.

[9] Apple Computer. *Inside Macintosh: More Macintosh Toolbox*. Addison-Wesley, 1993.

[10] Apple Computer. *Inside Macintosh: Text*. Addison-Wesley, 1993.

[11] Apple Computer. *Inside Macintosh: Files*. Addison-Wesley, 1994.

[12] Apple Computer. *Inside Macintosh: Imaging With QuickDraw*. Addison-Wesley, 1994.

[13] Apple Computer. *Inside Macintosh: Macintosh Toolbox Essentials*. Addison-Wesley, 1994.

[14] Apple Computer. *Inside Macintosh: Memory*. Addison-Wesley, 1994.

[15] Apple Computer. *Inside Macintosh: PowerPC System Software*. Addison-Wesley, 1994.

[16] Apple Computer. *Inside Macintosh: Processes*. Addison-Wesley, 1994.

[17] Apple Computer. *Inside Macintosh CD-ROM*. Apple Computer, Inc., 1995.

[18] Apple Computer. *Inside Macintosh: X-Ref*. Addison-Wesley, 1995.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[20] Rajeev Goré. Tableau methods for modal and temporal logics. Technical Report TR-ARP-15-95, Australian National University, 1997.

[21] Joseph Y. Halpern and David A. McAllester. Likelihood, probability, and knowledge. *Computational Intelligence*, 5:151–160, 1989.

[22] Joseph Y. Halpern and Michael O. Rabin. A logic to reason about likelihood. *Artificial Intelligence*, 32:379–405, 1987.

[23] Jan L. Harrington. *C ++ programming with CodeWarrior*. AP Professional, 1995.

[24] Jan L. Harrington. *CodeWarrior software development using PowerPlant*. AP Professional, 1996.

[25] W. Heinle. *Expressivity and Definability in Extended Modal Languages*. PhD thesis, TU München, Germany, 1995.

[26] A. Heuerding, G. Jäger, S. Schwendimann, and M. Seyfried. Propositional logics on the computer. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *Tableaux 95*, LNCS 918, pages 310–323, 1995.

[27] A. Heuerding and S. Schwendimann. On the modal logic K plus theories. In H. Kleine Büning, editor, *CSL 95*, LNCS 1092, pages 308–319, 1996.

[28] A. Heuerding, M. Seyfried, and H. Zimmermann. Efficient loop-check for backward proof search in some non-classical propositional logics. In P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, editors, *Tableaux 96*, LNAI 1071, pages 210–225, 1996.

[29] Alain Heuerding. *Sequent Calculi for Proof Search in Some Modal Logics*. PhD thesis, University of Bern, 1998.

[30] Alain Heuerding, Gerhard Jäger, Stefan Schwendimann, and Michael Seyfried. The logics workbench lwb: A snapshot. *Euromath Bulletin*, 2(1):177–186, 1996.

[31] Alain Heuerding, Michael Seyfried, and Heinrich Zimmermann. Efficient loop-check for backward proof search in some non-classical propositional logics. In Pierangelo Miglioli, Ugo Moscato, Daniele Mundici, and Mario Ornaghi, editors, *Theorem Proving with Analytic Tableaux and Related Methods*, number 1071 in LNAI, pages 210–225. Springer-Verlag, 1996.

[32] John Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, 1996.

[33] Stanley B. Lippman. *Inside the C++ Object Model*. Addison Wesley, 1996.

[34] Stanley B. Lippman and Josée Lajoie. *C++ Primer*. Addison-Wesley, third edition, 1998.

[35] J.-J. Meyer and W. van der Hoek. *Epistemic Logic for AI and Computer Science*, volume 41 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995. 25.5.103.

[36] Scott Meyers. *Effective C++*. Addison-Wesley, second edition, 1996.

[37] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996.

[38] The Opengroup. `http://www.opengroup.org/motif`, 1999.

[39] Developer Technical Publications. *Apple Event Registry: Standard Suites*. Apple Computer, Inc., 1992.

[40] Qt. `http://www.troll.no/products/qt.html`, 1999.

[41] D. Sahlin, T. Franzén, and S. Haridi. An intuitionistic predicate logic theorem prover. *Journal of Logic and Computation*, 2(5):619–656, 1992.

[42] Glen Sanford. Apple-history. `http://www.apple-history.com`, 1999.

[43] Stefan Schwendimann. *Aspects of Computational Logic*. PhD thesis, Universität Bern, 1998.

[44] Silicon Graphics Computer Systems. Standard template library programmer's guide. `http://www.sgi.com/Technology/STL`, 2000.

[45] Larry Wall and Randal L. Schwartz. *Programming perl*. O'Reilly & Associates, Inc., 1991.