# Aspects of Computational Logic

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Stefan Schwendimann

von Niederstocken (Be)

Leiter der Arbeit: Prof. Dr. Gerhard Jäger,
Institut für Informatik und angewandte Mathematik (IAM)

# Aspects of Computational Logic

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Stefan Schwendimann

von Niederstocken (Be)

Leiter der Arbeit: Prof. Dr. Gerhard Jäger,
Institut für Informatik und angewandte Mathematik (IAM)

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Der Dekan:
Prof. H. Bunke

Bern, den 2.4.1998

Aspects of Computational Logic


## Abstract

In mathematics there exist powerful symbolic computation packages which have drawn considerable attention, also because of their easy-to-use interfaces and graphical capabilities. In computer logic, however, there were mainly complex tools for experts or purely didactic proof assistants. The Logics Workbench LWB is an attempt to fill this gap in the area of propositional logics. On the one hand, the LWB is intended for being used as an educational tool, especially for non-classical logics, and on the other hand as a programmable logic platform for more experienced users. The present thesis comprises three major parts: a system design, an empirical study, and a theoretical contribution.

(a) Beside a general introduction to the LWB, we will present design aspects of its key components in the first part: the kernel, the parser, the programming language and the user interface. It is shown that the adopted solutions are adequate for doing logic on the computer. The design goal is to provide a convenient and flexible system, similar to modern computer algebra tools.

(b) In the second part, a prototypical extension of the LWB is presented. For a selection of modal logics the LWB has been adapted for *distributed* proof search in a network of workstations. We will show that by comparatively modest means a surprising performance increase can be achieved. The solution is a competitive approach which empirically proves that the speed-up potential is high, often super-linear.

(c) In the third, theoretical part, we will present a new compact calculus for a well-known variant of temporal logic. The new calculus for which we will give correctness and completeness proofs is similar to modal calculi and allows a 'canonical' extraction of models. While other approaches need a second phase to check the fulfillment of the so-called eventuality formulae, a decision procedure based on the new calculus has only one pass.

With the LWB we have a user-friendly computer logic system facilitating the access to logical formalisms. Its flexibility has supported the empirical study which yielded promising results in distributed non-classical proof search, as well as the theoretical investigation in propositional linear time logic which is especially suited for specifications and verifications of reactive, non-terminating systems.

# Contents

# Chapter 1

# Introduction

We will begin with a brief description of the Logics Workbench, its libraries and its central components. The LWB is also the basis for the second part of the thesis (distributed non-classical proof search), and it incorporates a decision procedure based on the calculus presented in the theory part. An introduction to these parts, however, can be found at the beginning of the respective chapters.

## 1.1 The Logics Workbench LWB — A Computer Logic System

The Logics Workbench LWB is an interactive system allowing easy access to and easy manipulation of a variety of logical formalisms at the propositional level. The LWB is an integrated logic tool which is centered around a comparatively small kernel and which provides separate libraries for the most important propositional calculi. Here we will give a survey of the kernel and the libraries together with further central components of the LWB (see also Fig. 1.1). A comprehensive discussion of the system design will follow in the first part of the thesis.

Since the LWB is not exclusively intended for specialists, a great deal of effort has been spent on a comprehensive information system. This information system is also crucial in connection with one of the main applications of the LWB: it is a useful tool in training and education where it contributes to a better understanding of the interdependencies of logical systems and of the procedural aspects of various logical calculi.

### 1.1.1 The Structure of the LWB

Very schematically, the LWB comprises four main components which closely interact with each other: kernel and parser, logic modules, graphical user interface, and information system.

The LWB kernel provides the central data structures on which both the parser

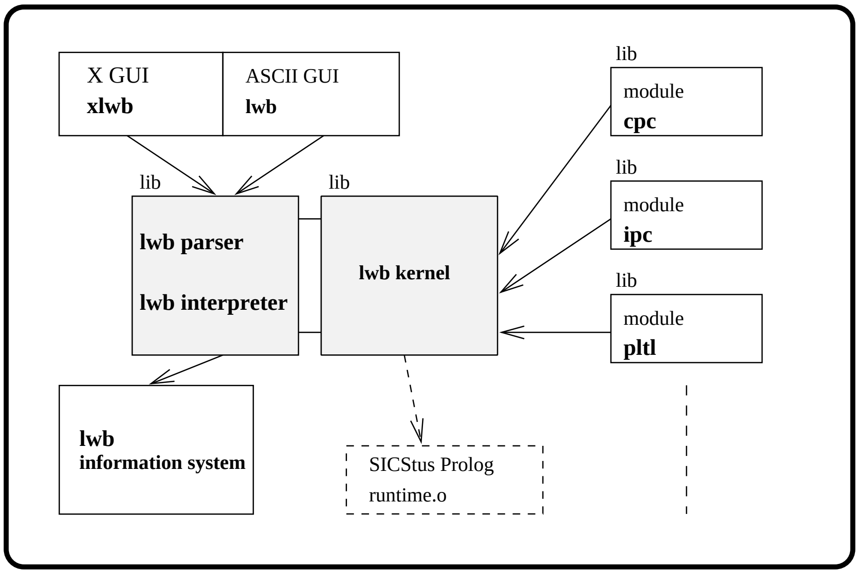



Figure 1.1: The modular structure of the LWB.

and the logic modules rely. The logic modules, however, contain a coherent collection of efficient implementations of well-known algorithms of the most important propositional logics:

1. Classical and intuitionistic logic.

2. Modal and multi-modal logics.

3. Temporal logics.

4. Non-monotonic logics.

5. Linear logic.

Various features ensure that users are able to interactively explore the system and the contained logic modules without further aids or lengthy preparation. The basic interaction with the system takes place via an untyped LWB language which follows very closely the philosophy of well-established computer algebra systems such as Maple and Mathematica.

A graphical user interface has been developed to facilitate the logging of sessions, the navigation within the program and the re-use of previously typed statements. A comprehensive information system gives detailed support for the use of the different modules. It contains uniformly grouped information and general introductions for selected logics. All information is available via the World Wide Web and may therefore be consulted without any local installations.

The structure of the LWB is modular. It allows the user to build a personal LWB according to his or her preferences and interests. It is even possible to link the kernel and selected modules to a third program and to use the LWB in this way as a background logic engine.

### 1.1.2 Kernel and Parser

The propositional formulae and other expressions that are used for algorithms are represented in the kernel of the LWB. The principles of object-oriented design have been applied. Great care was taken to provide the developers of LWB modules with a clear, abstracted interface to the basic objects. C++ allows good encapsulation, e.g. pointer handling becomes superfluous for the implementation of algorithms. Moreover, we took advantage of the possibility of operator overloading which makes dealing with expressions more intuitive and increases the readability of functions.

Great importance has been attached to a clear design of the internal class hierarchy and high efficiency of the basic member functions. The addition of new data types is simple; thanks to the inheritance mechanism of classes, only those parts that really differ from the other data types need to be implemented. In addition, there is a close relationship to the parser: the nonterminals of the grammar of the parser correspond to the abstract base classes, and the terminals correspond to the derived concrete classes. Reference counting has been chosen as internal memory management.

The user communicates with the system via the LWB language. This language is untyped (all objects are LWB–expressions) and is interpreted. It contains a fully–featured Pascal-like programming language which provides the user with all necessary constructs for rapid experimenting. It supports re-entrance (recursion), call by value and call by reference, lexical scoping, control structures (loops), etc. Although interpreted, the programming language is sufficiently efficient. The context-free grammar of the accepted language is given in Backus-Naur Form (BNF) out of which a parser is generated using the well-known tools lex and yacc.

### 1.1.3 Logic Modules

Each logic module provides a series of auxiliary functions (e.g. length and depth of formulae or proofs) and standard functions for the computation of various normal forms and the like. In addition, there are tools to compare different calculi and to compute embeddings and reductions.

For each logic 'random formulae' can be generated, allowing thorough functionality and efficiency tests of built-in (or user-added) algorithms.

The largest group of implemented modules concerns monotonic propositional logics:

1. Classical and intuitionistic propositional logic.
2. Modal propositional logics K, KT, S4, S5.
3. Provability logic.

4. Multi-modal logics $\mathsf{KT}_n$, $\mathsf{S4}_n$, $\mathsf{S5}_n$.

5. Tense logics.

6. Linear logic.

**Decision Procedures**

The key algorithm within each LWB logic module is the decision procedure. Most of these decision procedures for monotonic systems are based on two-sided sequent calculi. The standard forms of these calculi have been adapted and combined with several methods to improve the runtime complexity.

For efficiency reasons a specifically tailored decision procedure is used for each logic. If a generic algorithm were used, which would 'interpret' an abstract calculus description, it would be much harder to take advantage of specific knowledge about the logic under consideration. For instance, a costly loop checking is needed for some logics, whereas in others it can be avoided. Equally, embeddings are not suited to reduce the number of decision procedures since they often cross complexity boundaries. If we embedded, for instance, the classical propositional logic, which has an NP-complete decision problem, into intuitionistic propositional logic, we had to deal with a PSPACE-complete decision problem.

Besides pure derivability, the LWB is also able to cope with provability of a formula in a logic plus theory, that is, in a logic plus additional non-logical axioms. For classical and intuitionistic logic it is quite trivial (deduction theorem), but for several modal logics it requires some effort.

The module for classical logic contains an exemplary implementation of an algorithm which actually stores proofs in appropriate structures. Great care was taken to obtain *readable* results, e.g. by precisely keeping track of formula positions within a sequent. Quite some effort has been put into simplifying a found proof with *linear* cost. A graphical proof display tool called LWBproof_wish supports easy browsing across proof trees.

### 1.1.4 Graphical User Interface (GUI)

The GUI is based on X11 and Motif and is therefore called xlwb. It contains 3 components (see also Fig. 1.2):

1. Main window: the main window consists of a menu bar and a scrollable input window, which can be altered in size. The input window is the essential part of the GUI. It is built up using pairs of input and output regions. By returning to previous input regions, the user can save repeated typing of equal or similar commands. Each input region is a little multi-line text editor in which all needed text manipulations can be done.

2. Status window: it displays general information about the state and the activity of the LWB and specific information about particular runs of the algorithms. The user can control the amount of information according to his interest in details.

3. Option window: in the option window the user can change global variables such as display mode or bracket mode by mouse clicks. Likewise, he or she can choose his or her favorite color settings.

The version 1.0 supports various useful features:

1. Save/Load of all assigned variables, or the list of input and output regions (session).

2. Insertion and deletion of regions. Hiding the output regions. Re-evaluation of all input regions.

3. Emacs-like control sequences.

4. Connection to the help system (via help button).

The GUI has been developed under *OSF/Motif Release 1.2.2*. *Motif* has become a widely accepted standard and so the future support and the portability to other platforms are guaranteed. Special emphasis has been put on the similarity to other X11-/Motif-applications. For example, user specific adaptions are done via X resources.

### 1.1.5 LWB Information System

The LWB information system goes far beyond what is usually offered by other systems of computer assisted logics and mathematics. It relies on the features provided by World Wide Web (WWW). This approach offers several advantages:

1. Availability: any WWW browser (e.g. Netscape) is sufficient to use the LWB information system installed in Bern. If it is intensively used, or if the connection is slow, the users can also install it at their sites.

2. Representation: the rich Hypertext Markup Language (HTML) format allows a visually attractive representation using fonts in different shapes and sizes, and also icons and pictures.

3. Hypertext links: the LWB information system uses the hypertext capabilities to structure the available information.

4. Flexibility: an important advantage is the easy extendibility. Existing and upcoming Web techniques can be incorporated as required.

The LWB information system also contains a large variety of information besides the standard help texts:

1. Sample session with the LWB, picture of a sample session with the xlwb.
2. Syntax description.
3. Description of each function (with examples), list of the functions in a module, list of the modules.
4. Reference manual (Postscript).
5. Theory.
6. Information about the LWB: Aims, news, planned work, installation instructions.

A key feature of the Web-based information system is the possibility to use the LWB itself via WWW. The user can compose queries in a HTML form and send them off. The results are computed by the LWB installation in Bern and returned as a new HTML page. This facility is especially useful for occasional queries, because no preceding local installation of the LWB or of the LWB information system is required.

The theory section contains a selection of basic calculi, definitions and theorems, illustrated and motivated by typical examples. In order to explain the chosen algorithms, several implementation-oriented variants of the calculi are introduced.

Entry point to the LWB information system is the LWB home-page (see also Fig. 1.2),

`http://lwbwww.unibe.ch:8080/LWBinfo.html`

All you need is an Internet connection and a WWW browser ...

## Acknowledgments

As with any significant work, there are a number of people who require special thanks. In particular, I would like to express my thanks to Prof. G. Jäger for the encouraging and motivating guidance. He has also initiated the LWB project. Equally, acknowledgment is due to Wolfgang Heinle for his constructive criticism in the part on propositional linear time logic, as well as to Rajeev Goré for many comments and his help in English formulations. Special mention is also due to the master's students who contributed to the LWB with their theses. The collaboration and discussions with them, and of course with Alain Heuerding, who worked on the same project, were most fruitful and contributed to the friendly atmosphere in the research group.

### 1.1.6 A Snapshot

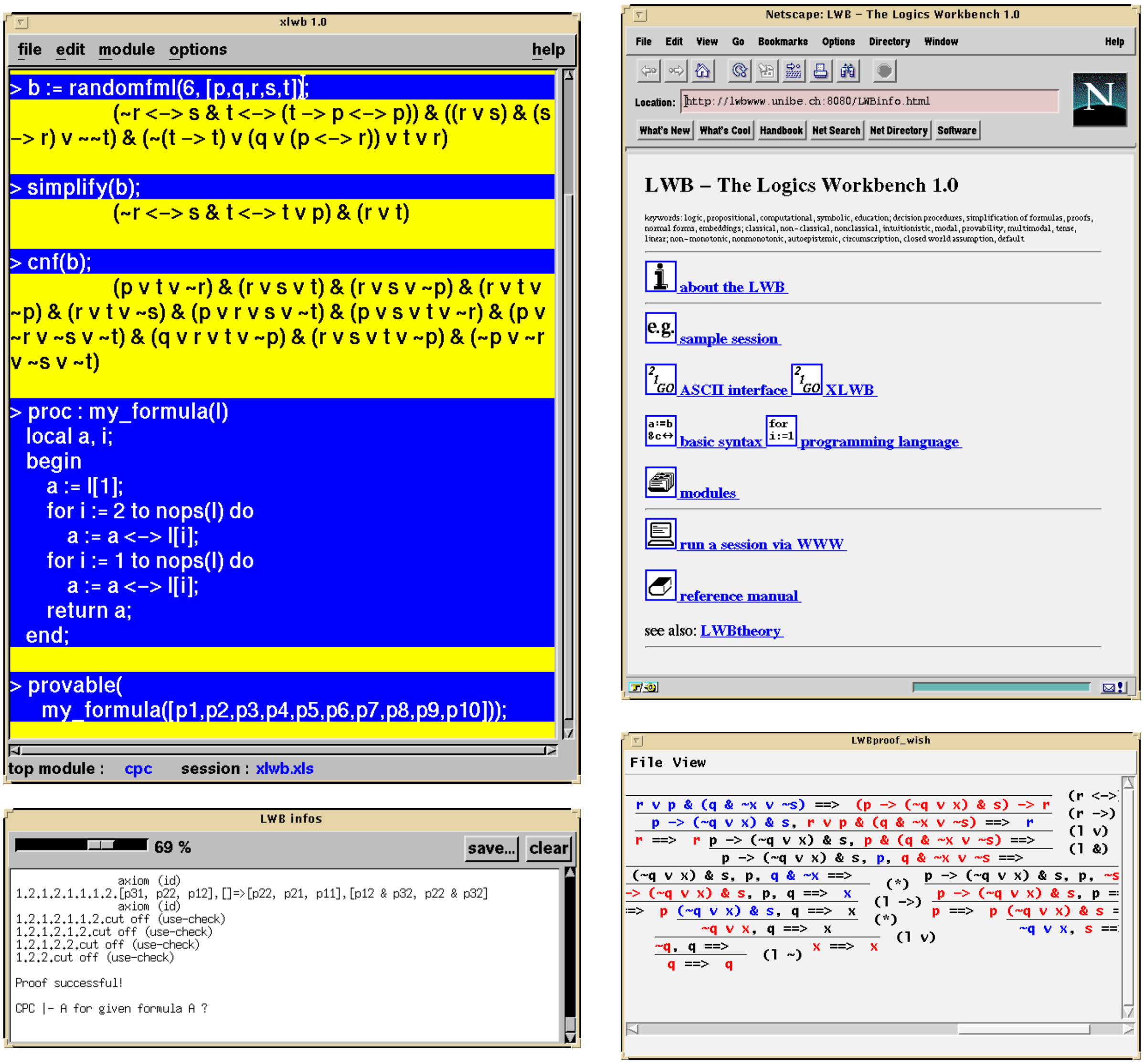



Figure 1.2: A snapshot of the Logics Workbench.

## 1.2 The Overall Design

### 1.2.1 Aims

In a preliminary report the Committee on Logic and Education of the Association of Symbolic Logic stated:

> *'In mathematics, symbolic computation packages like* Mathematica *and* Maple *are serious tools used for doing mathematics. There is nothing like that yet in logic.'*

The LWB is a first step in this direction. We claim that it is a *serious* tool in a limited area. In [58], for example, the LWB decision procedures and programming facilities were heavily used to find instances of formulae which relate variants of modal logics to each other. Efforts have been made to extend the LWB to meet the following requirements:

- User friendliness: Logic is probably even more abstract than other parts of mathematics and people are certainly less used to it. For such a system ease of access is most important and therefore an important aim was to make the LWB as easy to use as one of the popular computer algebra systems. Many other logic systems are very complex and notoriously difficult to learn how to use them.

- Efficiency: In order to cope with reasonable problems beyond the usual toy examples presented in logic courses, the algorithms and the underlying kernel must be efficient. Therefore, efficiency was a major concern in the design of the central data structures and the general layout of the logical algorithms. Furthermore, the interpreted LWB programming language is also remarkably efficient.

- Portability: To be accepted by the wider public, it is essential for a system to be portable (and to be ported) to the most important platforms. Portability has influenced the selection of tools and libraries used to build the LWB. Programs written in the LWB programming language are trivially portable. Currently, a MacIntosh and a Linux version are being developed.

- Stability: We have tried to provide a system which is stable also under incorrect manipulation. Meaningful error messages are given wherever possible and the erroneous position is precisely located. To guarantee the stability of the kernel and the algorithms, countless tests have been collected which can be executed automatically.

The LWB is a publicly available, stable system which can serve as a general platform for experiments in (propositional) logics. Moreover, it is not just a loose collection of algorithms but a comprehensive system providing a uniform environment to compare logics. The focus is mainly on the syntactical part (provability, proofs), and less on the semantic side (models).

### 1.2.2 Focus on Propositional Logics

The LWB is restricted to propositional logics for various reasons.

- Except for linear logic, all propositional fragments of the considered logics are *decidable*. This allows for fully automated proof search. Furthermore, the user can do much more than simply learn to prove theorems (which most logic systems teach). In the early stages, it is often more interesting to see what can be expressed in a particular logic, and for this purpose an automatic prover is crucial.
- The characteristic properties of a logic can already be observed in the propositional fragment. The step to first order is then very much the same in the different logics.
- Propositional logics are obviously less technical in that we do not have to care about bound variables or side conditions on occurrences of variables in rules. This is certainly an advantage for teaching.
- The decidability problems for propositional logics are definitely 'hard enough'. They belong to the most difficult problems (in the sense of complexity) of computer science. They range from NP-completeness (classical logic) over PSPACE-completeness (many modal logics) to EXPTIME-completeness (branching time logics) or even worse.

There are other reasons for the decision for propositional logics. The fact that most first order logics are not decidable usually forces the user to interact with the system when searching for proofs. And at the beginning an inexperienced user is unlikely to be a good guide. It is often demanded too much of a beginner to decide which direction he can take in a calculus he barely knows.

Another difficulty is that the formalizations of problems in first order logics often require reasoning about the natural numbers using the associated axioms and rules of induction. This is beyond the scope of the LWB.

It is, however, important to point out that the restriction is by choice. The self-imposed limitations are in no way essential. The inherent design of the system easily allows first order constructs to be incorporated as well as data structures for semantic functions.

### 1.2.3 Comparisons

Logic systems may roughly be divided into logic tools, proof development systems and theorem provers. The distinction is often not sharp. This comparison concentrates on the first family where the logic itself is the main issue. The LWB also fits in this group. Here we restrict ourselves to a selection of still existing and available systems. A database of mechanized reasoning systems is maintained at

`http://www-formal.stanford.edu/pub/clt/ARS/systems.html`

### Jape

Jape ('Just another proof editor') [8, 7] is an interactive, user-adaptable calculator for formal proofs written by Richard Bornat (QMW) and Bernard Sufrin (Oxford). Its emphasis is on a well-designed graphical interface and configurability by the user. Jape includes a straightforward implementation of first order logic (natural deduction, sequent calculi), equational reasoning and various special applications, for example, authentication protocol verification. Unfortunately, modal logics and linear logics are not supported (yet). Jape is a typical representative of a proof development tool and is a good complementary system to the LWB, since in Jape very little is done automatically. An advantage with respect to other systems is that syntax, rules and the corresponding user interface facilities are configurable. New logics can be encoded in this way.

`ftp://ftp.dcs.qmw.ac.uk/programming/jape/index.html`

### Mac Logic

Mac Logic [20] is another well-known proof assistant developed by R. Dyckhoff and others at St. Andrews University. It covers classical and intuitionistic logic and two modal logics (S4 and S5). It has two modes, one for proof checking and one for proof construction. It supports teaching top-down first order theorem proving using Gentzen's sequent calculi LK and LJ or natural deduction. When doing first order proofs, automatic theorem provers for propositional subproblems can be activated so that a user can concentrate on the first order difficulties. What is there is sound and well done (it is also a commercial product), but it lacks the flexibility of Jape.

`http://www-theory.dcs.st-and.ac.uk/~rd/logic/intro.html`

### Hyperproof

Hyperproof [5] is a special system by J. Barwise and J. Etchemendy to teach formal deduction. It is designed to be used in introductory logic courses. The speciality is that it combines graphical information (a blocks world situation) and sentential information. The idea is that the heterogeneity of 'visual' reasoning and syntactical reasoning reflects better everyday situations. It also alludes to the tradition of diagrammatic methods (Euler, Venn, Peirce). Students can also learn to establish non-consequence. Hyperproof is a successor system of Tarski's World by the same authors.

`http://csli-www.stanford.edu/hp/Logic-software.html`

### ILFA

ILFA[1] [19] is an experimental logic computation system developed by the group of Prof. Kleine-Büning at the University of Paderborn. It is essentially a large reusable

[1] It will be renamed to logicAL.

library of logic algorithms for classical propositional and first order logic. There exist specialized provers for certain classes of formulae, e.g. propositional Horn formulae, and many variants of different proof methods (e.g. resolution strategies), which can be compared with each other. Specialized data structures are used to take advantage of the special form of the formulae. The next version of the LWB will incorporate a selected subset of the classical algorithms of ILFA, especially an implementation of the Davis-Putnam decision procedure which has turned out to be the most efficient proof method for classical propositional logic.

`http://www.uni-paderborn.de/fachbereich/AG/agklbue/staff/lettmann/`

### Proof Development Systems

There are many other logic systems out there. For instance Lego [52], also a natural deduction proof assistant with support for type systems, $\lambda$-calculus and inductive data types. Nuprl [27] is a big proof development system which can be viewed as the implementation of Bishop's ideas on constructive mathematics. Isabelle [57] is a widespread generic theorem prover which allows to specify syntax, rules and proof procedures (using tactics and tacticals). Coq [43] is a package which supports type theory and its application to software verification. These proof development systems are strongly oriented towards mathematical formalisms and require a considerable amount of time to learn how to use them.

### Theorem Provers

The Boyer Moore Theorem Prover (BMTP) [10] is one of the most powerful automated reasoning systems dealing with computationally definable objects represented as Lisp functions. It can apply induction schemes to recursively defined data types and functions. Otter [67] is one of the most important resolution-based theorem provers for first order logic with equality. SETHEO ('SEquential THEOrem prover') [48] is a top-down theorem prover of similar performance based on the so-called connection tableau method. There exist also parallel versions called PARTHEO [62] and SPTHEO [63]. One of the more specialized provers is for instance 3TAP [6] which can deal with many-valued first order logic with equality.

Most of these theorem provers are designed to solve large problems in classical first order logic and also require a good knowledge of the system and the method options to be used efficiently.

## 1.3 Scope of the Thesis

### 1.3.1 About the Contents

There are 3 parts:

- System design part: in this first part, the design of the LWB is explained. From the user's point of view the design of the LWB input language is the essential part. We might call this the 'external' design because the system is mainly perceived by the external world through the language. We will point out why we think that the LWB input language is appropriate for computational logic. As next step the 'internal' design is presented. The key components of the system are discussed, and it is shown how they interact with each other. Of course, the internal design has also influenced the design of the input language when efficiency or elegance of implementations were affected. Implementation details, which should ease the future maintenance, can be found in a technical appendix.

- Empirical part: we extended the existing LWB into a prototype for distributed proof search in a UNIX environment. We show that the necessary changes are modest and that already a small number of machines can increase the performance enormously; often a super-linear speed-up can be observed in non-classical proof search. On the theoretical side, the modifications can be captured by a suitable adaption of the calculi.

- Theoretical part: presumably, temporal logics have the greatest range of practical applications and their relevance for computer science (verification of hard- and software) is obvious. In this part, we will present a new calculus for the propositional linear time logic PLTL and give correctness and completeness proofs for it. The new calculus can serve as a basis for a one-pass decision procedure similar to those in modal logics. An apparent advantage is that linear counter models, which are needed for 'debugging' purposes in verification, can be extracted in a canonical way. Of course, there is a module in the LWB for PLTL with an implementation.

### 1.3.2 Related Work

The work described here is closely related to the thesis of Alain Heuerding who is co-author of the Logics Workbench. While his main interest is in modal logics (proof techniques and related topics), the focus here is on the general aspects of building a system for computational logic. We refer also to the thesis of Michael Seyfried where a new calculus for the $(\rightarrow, \wedge)$–fragment of intuitionistic propositional logic is presented. He also shows, how loop-checking, which is needed to ensure termination, can be incorporated in the definition of the calculus.

- A. Heuerding. *Sequent Calculi for Proof Search in Some Modal Logics.* PhD thesis, IAM, University of Bern, 1998.
- M. Seyfried. *Computational Aspects of Fragments of Intuitionistic Propositional Logic.* PhD thesis, IAM, University of Bern, 1998.

More information on design and technical realization of parts of the LWB is available in the diploma theses of Franz Achermann [1] and Peppo Brambilla [11]:

- F. Achermann. The LWB programming language. Master's thesis, IAM, University of Bern, 1995.
- P. Brambilla. Sequent calculus proof representation. Master's thesis, IAM, University of Bern, 1996.

### 1.3.3 Class and Object Diagrams

In the system design part, diagrams are used to illustrate the relationship of different classes and objects. The style of the diagrams is according to the Unified Modeling Language (UML). See appendix A.1.2 for a short description of the employed design elements of UML.

### 1.3.4 Typography

Within text paragraphs the following typographical conventions are applied:

| | font | example |
|---|---|---|
| variables (LWB, C++) | typewriter | `myvar` |
| literal statements (LWB,C++) | typewriter | `for(int i=0; i<10; i++);` |
| meta-statements | slanted | *loop over A* |
| literal input (LWB) | typewriter | `load(cpc);` |
| meta-input | slanted | *load appropriate module* |
| tools | sans serif | perl |
| filenames | sans serif | foo.C |

Function names ( LWB programming language, C++) are always followed by a pair of parentheses, e.g. `f()`, omitting the argument list for brevity. The UNIX-like notation foo.[hC] for files is a short cut for 'foo.h and foo.C'.

# Chapter 2

# The LWB Kernel

The comparatively small kernel is the key component of the Logics Workbench. All other elements of the LWB are centered around this kernel. However, the kernel is independent of the other parts.[1] Figure 2.1 shows a survey including the relations to the parser, the user interface(s) and the logic modules.

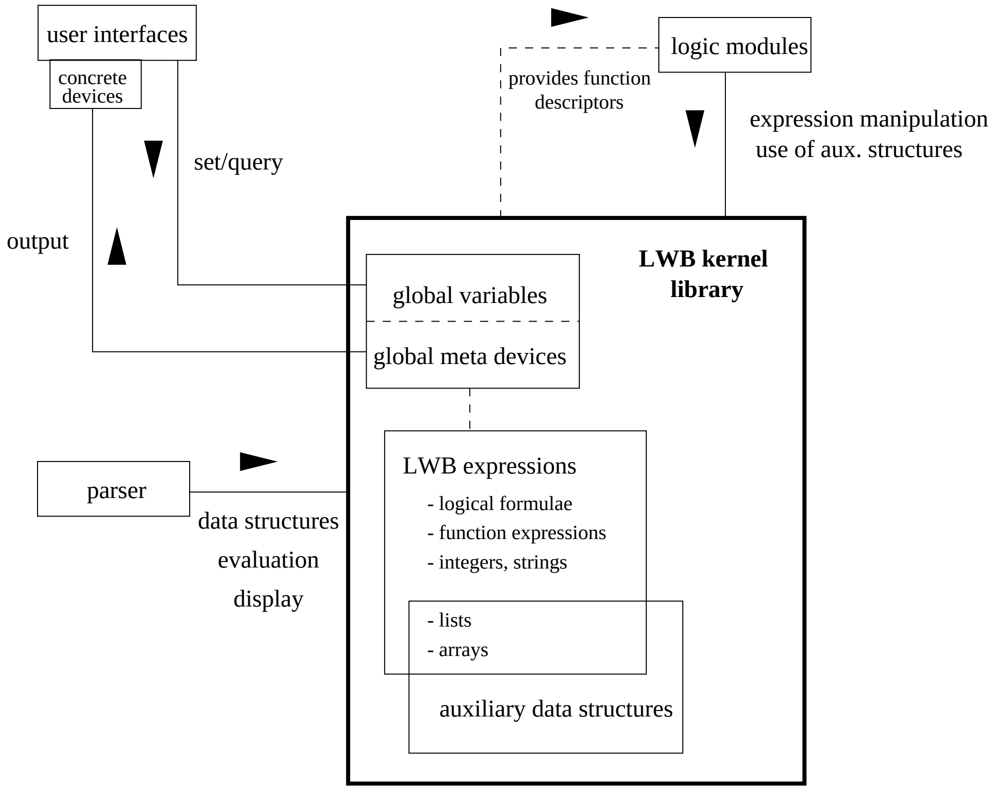



Figure 2.1: The LWB kernel library.

The main purpose of the kernel is to provide an implementation of the central data

[1] There is an exception. The meta-devices for basic output must be assigned to real output devices, which is usually done in the user interface.

structures that the user works with in the LWB input language. From the user's perspective, everything is an *expression* to which a uniform set of operations can be applied. Moreover, since no declarations are needed except for local variables inside a procedure, the language appears to be 'untyped', or equivalently, the only type is 'expression'. Of course, there are different kinds of expressions, for instance, logical formulae, lists, strings etc., which are solely discerned by a different syntax. So each expression is of some 'subtype' which a user can also query explicitly using the `type()` command. The subtype is used to analyze expressions in procedures written in the LWB input language.

```
> type( a & b );
        AND
> type( [a,b] );
        LIST
```

On the implementation side, this concept is naturally mirrored by an object-oriented design where we have an abstract class at the top and different levels of derived subclasses. Each instance of a specific subclass also belongs to the superclasses in the hierarchy above and can especially figure as an instance of the top class, in our case as an LWB expression.

The following tasks and components of the LWB kernel are discussed in this chapter:

- The hierarchy of LWB expressions, of which the most important ones are the logical operators. For instance, the conjunction, whose input language symbol is `&`, is implemented as derived class `andop` (cf p. 24).
- The auxiliary data structures as expression stacks, lists, dynamic arrays, etc. They are used in all logic modules (cf p. 38).
- A small set of LWB-global variables (e.g. a global LWB-expression representing `true` (cf p. 27).
- The evaluation strategy for global symbols (cf p. 25).
- The basic output of data structures and the concept to direct it to different user interfaces (cf p. 29).

Note that at runtime the kernel stores no data except for the few global variables; the kernel provides mainly services (implementations). Expressions (instances) are stored at other places:

- The symboltables of the modules maintain expressions. In particular, the user module stores the user-defined formulae.
- The runtime stack of the LWB interpreter references expressions as intermediate results, local variables, etc.
- The module functions create and delete auxiliary structures containing expressions.

# 2.1 The LWB Expressions

## 2.1.1 Introduction

The LWB expressions, implemented as class `expr`, are the most important data type of the LWB. They are the basic data structure which the user deals with to express and manipulate logical formulae, and also the functions of the logic modules work on expressions. A typical scenario is the following:

> User input:
>
> `> provable( a & b -> a );`
>
> > The interpreter/kernel:
> > The interpreter constructs expressions using kernel functions. To obtain *evaluated* expressions, the `evaluate()` method of `expr` is called, here for the expressions corresponding to the basic symbols `a` and `b`. The `provable` symbol evaluates to a function descriptor which is also an expression subtype. The descriptor points to a module function.
> >
> > > The module functions:
> > > The module functions proceed often by a recursive case analysis; the `type()` method of `expr` indicates the specific concrete subclass and the subscript operator allows to decompose the expressions step by step. The result of a decision procedure is normally one of the global kernel expressions `trueexpr` or `falseexpr`.
> >
> > The interpreter sends the result expression to the global output device.
>
> Result at the user level:
> `true`

## 2.1.2 The Public Interface of the Expression Class

### Creation

The symbols for atomic formulae, the propositional variables, are usually controlled by the symboltables and are not directly created by algorithms. When a symboltable is asked to create a symbol, it first checks to see whether the symbols exist already. If not, it uses the `expr` constructor

```
expr::expr(char *s);
```

to create a concrete instance of the subclass `symbol`. On the other hand, to compose new formulae out of existing ones, the kernel provides for each logical operator a function, whose name is similar to the verbose version in the input language.[2] Of course, the C syntax has to be observed now, that is, new formulae must be defined in a (functional) prefix notation. For example, the formula of the above scenario could directly be defined in the kernel by the command

```
imp( and(a,b), a )
```

when we assume that `a` and `b` are of type `expr`.

### A Uniform Set of Operations

a) A group of operations has an immediate relation to the user syntax. Beside simple manipulations at the user level, this group is also used for writing algorithms; either in the interpreted input language or directly in the kernel as the logic modules do.

| | | |
|---|---|---|
| 1. | Type of top-level operator : | `int type() const` |
| 2. | Number of operands : | `int nops() const` |
| 3. | Subscript [1 ... `nops()`] : | `expr& operator[] (int i) const` |
| 4. | Subscript [0] : | `expr& index0 () const` |
| 5. | Substitution : | `expr subs(const fixedlist &S)` |

b) Two operations are used for the language interpretation:

| | | |
|---|---|---|
| 1. | Evaluation : | `expr evaluate()` |
| 2. | Explicit copy : | `expr copy()` |

While evaluation is used to obtain the values of metavariables, the explicit copy method is called to split a shared expression when an input language operation is changing the expression itself.

c) A set of output member functions: They send the expression in different formats on so-called devices. See the section on the LWB output 2.3.

### The Orthodox Canonical Class Form

If we want to use instances of the class `expr` as any other C variable, the class must supply a set of methods which guarantees that a variable can be assigned, declared and passed as an argument.

- The default constructor `expr::expr()`: If no initialization context is provided, the default constructor initializes the `expr` instance to a default value. This constructor is called by the compiler to initialize the members of an `expr` array.

- The copy constructor `expr::expr(const expr &e)`: This constructor is called by the compiler when an `expr` is passed by value, and sometimes when one is returned as a value. It should build an exact copy of the given expression `e`. Actually, it copies the given expression only logically (see the following section on memory management).

[2] See file expr.h.

- An assignment operator `expr & expr::operator=(const expr&)`: In an assignment we have to explicitly clean up the old value before assigning the new one.

- The destructor `expr::~expr()`: The destructor must undo the allocations made by an `expr` constructor. It is for instance activated when an explicit `delete` deallocates an array of expressions or when the scope is left in which an `expr` variable has been declared.

If a class implements these four methods, it is also said to be in 'orthodox canonical form' [14]. Another basic method is certainly the equality operator

```
int expr::operator==(const expr &f) const;
```

that allows the comparison of two `expr` instances. For a complete list of methods see the code fragment 1 in A.2.1.

### 2.1.3 The Memory Management

LWB expressions are often assigned and passed as arguments. A simplistic implementation of the corresponding methods (assignment operator, copy constructor) would cause entire expressions to be replicated, thereby almost doubling the memory requirements. So a fundamental internal design decision is the choice of a suitable memory management which avoids useless redundancy.

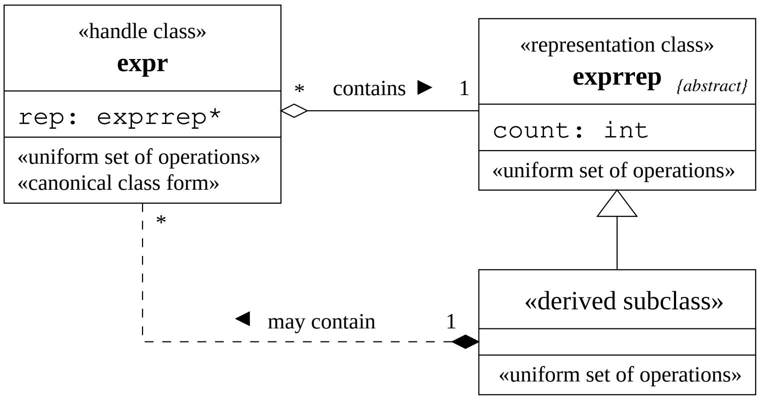



Figure 2.2: Reference counting and derivation scheme for expressions.

Particularly adequate for expressions, which dynamically allocate memory, is the *reference counting* scheme. The LWB expressions are split in two classes: a handle class `expr` which serves as the programming interface (see the section above) and a representation class `exprrep` in which the expression 'intelligence' lies. The idea is that instances of the handle class can share instances of the representation class which store the number of referring handles in a reference count. The class diagram 2.2 shows the reference scheme and a possible relation of a derived class to the handle class. Note that `exprrep` need not be in canonical class form since `exprrep` instances are supposed to be accessed via `expr`.

The methods of the uniform set of operations are simply forwarded from `expr` to their counterparts in `exprrep`. The second role of the handle class is to perform the memory management for `exprrep`. Whenever possible, the expression representations are shared, that is, expressions are only *logically* allocated and deallocated. In the object diagram 2.3 the situation before and after the `expr` assignment `a = b` is shown. The count of the representation of `b` is incremented, and the count of the representation of `a` is decremented. If $m-1$ were 0, then the latter would also *physically* be deleted.

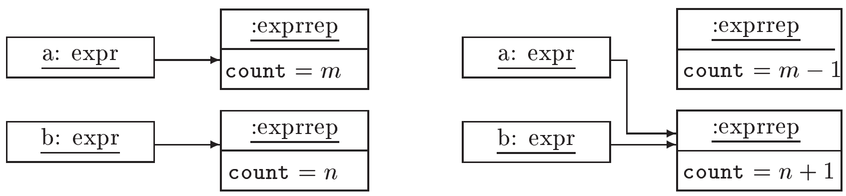



Figure 2.3: Before and after the `expr` assignment `a = b`.

Reference counting is just one form of garbage collection. Its advantage is that the current state of allocations is precisely captured by the reference counts. Whenever a count reaches zero, the storage is immediately freed. This is important when memory is short or when algorithms repeatedly allocate and deallocate large structures. Other garbage collection techniques such as mark–and–sweep perform an automatic and periodic storage reclamation. It is usually triggered during memory allocation when the amount of free memory falls below some threshold or after a certain number of allocations. So the garbage collector interrupts occasionally the execution of algorithms which may lead to different execution times. Both reference counting and periodic garbage collection are viable solutions for a symbolic computation system. While periodic garbage collection is usually done for all allocations, the reference counting is often restricted to certain types of objects (here expressions) and aims directly at structure sharing.

### Expression Trees

Expressions can be represented as trees. Note that each node (operator) is reference counted and can be shared. Figure 2.4 shows an object diagram of a conjunction; on the left in the detailed UML style and on the right a simplified representation.

### Recursive Traversals and Structure Sharing

Recursive functions traversing a formula to create a modified formula often proceed as follows: they look at the top-level operator, do a recursive call on the operands and build a new instance of the same operator using the results of the recursive calls as operands. This leads in many cases to unnecessary physical copies. Even if

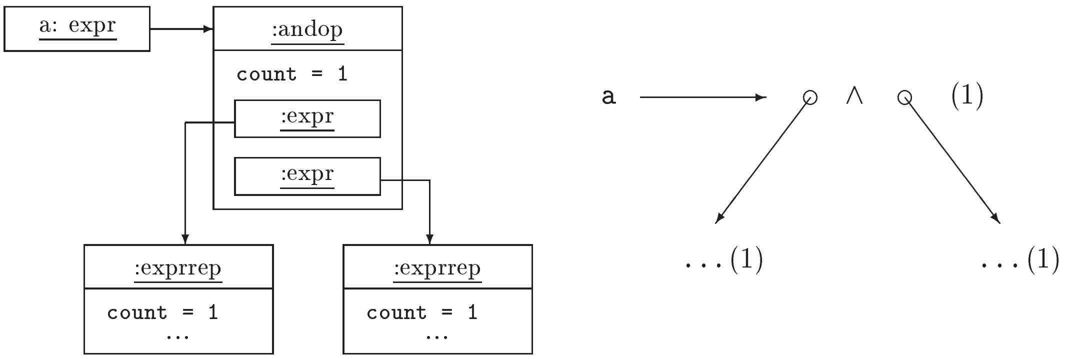



Figure 2.4: Object diagram (UML and simplified) of a conjunction.

the modification does not change the current subformula — i.e. the result formula could share the representation with the original subformula — the entire subformula is physically copied. Two important expression member function, the substitution and the evaluation, circumvent the disadvantage by the following modified scheme:

1. As before, the function is recursively applied to the operands of the object.
2. If each recursive call returns the operand itself, the function returns the current subformula (using the `this` pointer). The result is then a logical copy.
3. If, however, one operand has changed, a new instance of the current operator with the new operands is returned. The operands which have not changed are still shared between the original subformula and the result.

In this way, the modified formula and the original formula share all equal subformulae being at the same position. Figure 2.5 shows three formulae where the middle and the right formula are obtained by applying substitutions to the left formula. In fact, the middle formula shares the representation with the left formula because the substitution has no effect.

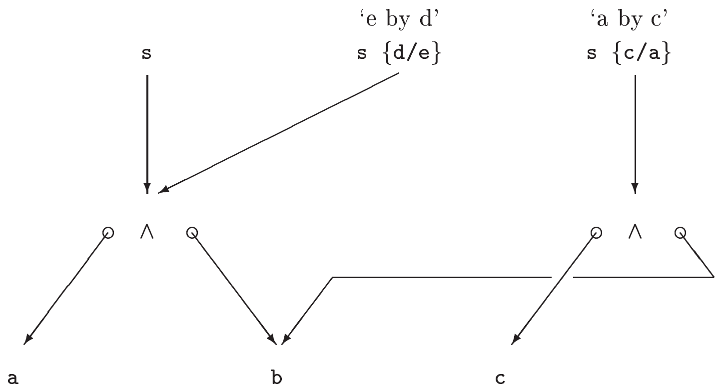



Figure 2.5: Example for substitution.

### Side-Effects

While the benefits of structure sharing are obvious, there are also disadvantages, namely the imminent danger of unintended side-effects when operations on expressions are not purely functional. Consider the following example (on the left the input language syntax, on the right the corresponding kernel code):

| LWB syntax | C++ syntax |
|---|---|
| `> s:= a and b;` | `expr s = and( "a", "b" );` |
| `> t:= s or c;` | `expr t = or( s, "c" );` |

We obtain a partially shared structure as depicted in Fig. 2.6. Obviously, any change to an operand of `s` would also change `t`, and conversely, a change to the left operand of `t` would also change `s`. In the interpreted input language such side-effects can be avoided since assignments and argument passing are under the control of the interpreter. The solution is to check the reference count and to copy the nodes of the expression tree that are referenced more than once (see also 3.4.3).

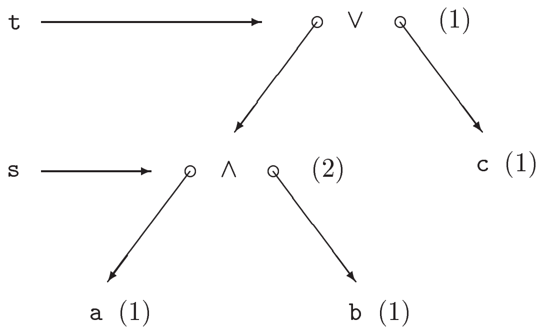



Figure 2.6: A shared expression.

The situation is different when we deal directly with the kernel with which not all kinds of argument passing can be controlled. In the following we will briefly address the caveats regarding expression argument passing. Note that the methods of the canonical class form take only care of the memory management, i.e. that unreferenced expressions are removed and that referenced expressions are not accidently deleted. However, they do not protect against side-effects.

### Passing Expressions as C++ Arguments

**Call by Value:** When an expression is passed by value, the copy constructor creates a logical copy and accordingly, when the procedure is left, the destructor does a logical deletion.

```
void f1 ( expr e )                                    // call by value
{
```

```
    // At entry the copy contructor increments e.rep->count
    ...
    // At exit the destructor decrements e.rep->count
}
```

The logical copy and deletion is very fast because they consist only of an increment and decrement of the reference count. On the other hand, we cannot simply change the argument as other value arguments since otherwise we have a side-effect. Therefore, the part of the expression that will be changed must still be copied physically and unfortunately explicitly using the explicit `copy()` method of `expr`. The untouched subexpressions can still be shared. Such a programming style is obviously awkward and furthermore directly depends on the memory management solution. Thus, the procedures of the LWB logic modules never pass expressions by value. It is simpler and more natural to declare a local expression and to initialize it with parts of the passed expression.

**Call by Reference:** To pass an expression by reference is most dangerous with respect to side-effects. Consider symbol `a` in Fig. 2.6. If it is passed as reference argument to a procedure, the count is still one, although it is part of two expression trees! Since there is no father relation, there is no way to find out whether a higher node in the expression tree is multiply referenced.

```
void f2 ( expr &e )                          //  call by reference
{
    // Modifications to parameter e change the actual argument.
}
```

This way of passing expressions is not used in logic module procedures and is not recommended.

**Call by Constant Reference:** A compromise between call by value and call by reference is the call by 'constant reference' (which is C++ specific). The runtime behavior is like the efficient call by reference. However, the compiler ensures that the argument is treated like a constant inside the procedure. That is, it may not be reassigned, and if it is an instance of a class, only methods declared to be `const` can be applied.

```
void f3 ( const expr &e )              // call by (const) reference
{
    // No modifications allowed by the compiler
}
```

This type of argument passing is recommended for expressions since it is safe and efficient. It requires the procedures to be written in a functional style (at least as regards expressions). However, this is no real limitation in our context. LISP for instance, which is the prototype of a functional language, has a long tradition in computational symbolic mathematics.

### 2.1.4 The Expression Hierarchy

In Fig. 2.2 the inheritance scheme is sketched. An expression subclass is derived from the abstract representation class `exprrep`. If a subclass contains several other expressions, we call it a 'composed expression'. The contained expressions are deliberately not declared to be of a specific subclass but are general expressions, which increases the flexibility of composition. To let the derived subclasses benefit from the services provided by the handle class, the contained expressions are of type `expr`. So at the class level, we have a circular dependency. Of course, circular dependencies must be avoided at the object level: an `expr` instance should not be contained by the expression tree that it references[3]. Otherwise, recursive tree traversals never end. The full derivation tree is shown in Fig. 2.7. Only single inheritance is employed.

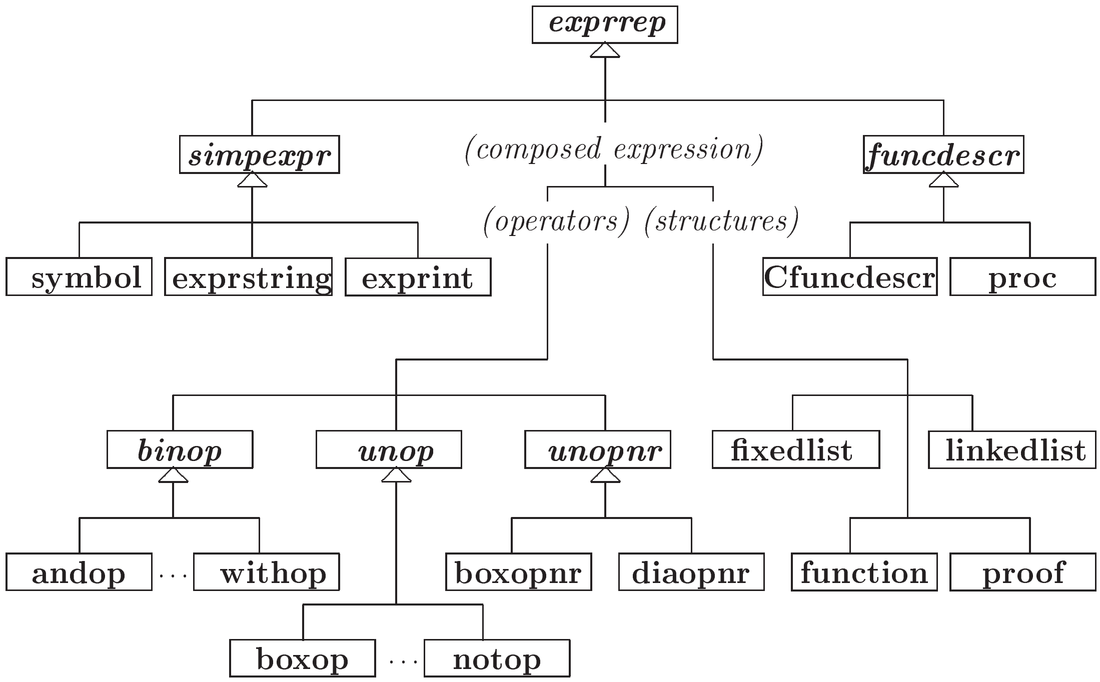



Figure 2.7: Expression class hierarchy.

Some groups of expressions have an intermediate superclass which is also abstract (class name in italics). A superclass is introduced if there is a sufficient 'procedural coincidence' so that the superclass can provide an actual service for its subclasses

[3]A circular dependency can only be created through assignments. The interpreter verifies input language assignments using the `exprrep` method `hasrep()` which checks whether the `rep` pointer of the symbol to which an expression is assigned is already contained in the expression tree.

(code sharing). There is no superclass if we have only a semantic coincidence, as for instance between the binary and unary operators which are both 'logical operators', but which have nothing else in common.

## 2.2 Logical Formulae

In the following we focus on the expression subclasses which are used to represent the logical formulae: symbols (they serve as propositional variables) and operators.

### 2.2.1 Symbols

As an expression subclass, symbols play a double role in the LWB. If they are unassigned, they stand for themselves. Unassigned symbols represent the propositional variables in a logical formula. But symbols can also stand for another formula, or more generally, for any other expression. In that case, they serve as metavariables. Before a formula is displayed or passed to an algorithm, the contained metavariables must be *evaluated*. As mentioned before, the symbols are normally managed by the symboltables of the logic modules and especially of the user module (see Fig. 2.9).

The symbol class is derived from the abstract class `simpexpr` (see fig. 2.8) which implements the part of the uniform set of operations that is equal for all atomic expressions. The symbol class contains as main data a character string `str`, the name of the symbol, and an expression `e`. If `e` is empty (uninitialized), the symbol is not a metavariable, and the `evaluate()` method returns the symbol itself.

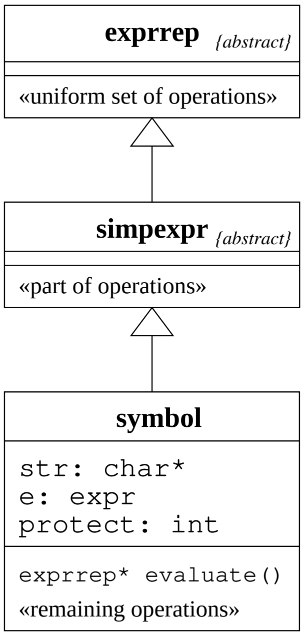



Figure 2.8: The symbol class.

#### Evaluation

Since a formula represented by a metavariable may again contain metavariables, the process of evaluation is recursive. The rules for evaluation are as follows:

1. Evaluation is only triggered by the interpreter to resolve the metavariables a user may have (implicitly) introduced.

2. Full evaluation down to unassigned symbols is performed on the right-hand-side (rhs) of an (input language) assignment. The evaluated rhs which is the value of the expression at that time is assigned to the lhs and contains therefore

no meta-variables. Later on, however, symbols of the rhs may be assigned in their turn, so that the structure pointed to by the lhs differs from its evaluated value (see also the example below).

3. Full evaluation is also performed when an expression is simply displayed. This is like an 'assignment without lhs'.

4. The call of a function in the input language is considered as the evaluation of the corresponding functional expression. Since the arguments are evaluated before they are passed to the actual function, the subsequent operations within algorithms never need further evaluation. The result expression of a function call, however, is not evaluated.

5. We explicitly have no evaluation in the following cases:

   (a) On the left-hand-side of an assignment.

   (b) The arguments of the `unassign`, the `protect` and the `unprotect` statement are not evaluated.

   (c) Local symbols within input language procedures are not evaluated.

   These cases are recognized by the parser and treated separately.

As an example, consider the following user statements:

```
> s:= a and t;
> t:= b;
> c or s;
```

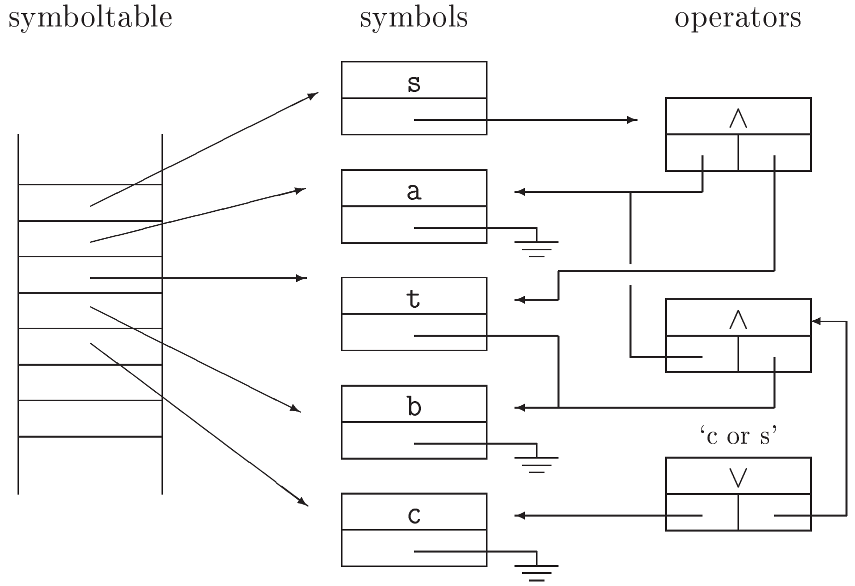



Figure 2.9: Symbols and Evaluation.

They induce a memory structure as shown in Fig. 2.9. For the third line a temporary expression is built up, with the evaluated `s` as left operand of the $\vee$-operator.

### 2.2.2 Operators

Logical operators are either binary or unary. For both types there is an intermediate abstract class, `binop` and `unop`, from which the concrete devices are derived.

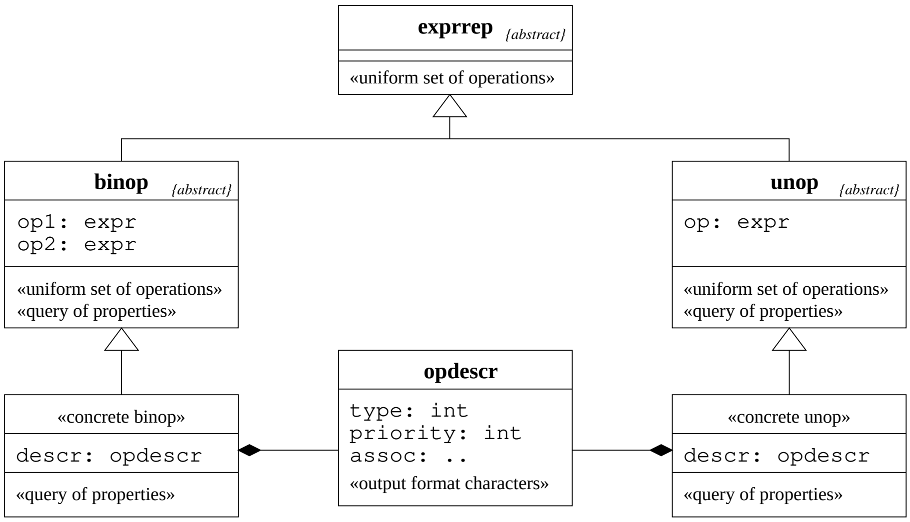



Figure 2.10: Binary and unary operators.

Since the various operators differ only in a couple of properties, e.g. their associativity or the way they are displayed, the intermediate classes can provide the required methods in terms of these properties. The derived classes contain an instance of the class `opdescr` in which the operator properties are stored.

### 2.2.3 Global Variables and Constants

In this section we present an overview of the global variables and constants of the LWB. There are different groups:

- User controlled variables for output features:
  - Bracket mode (`bracketmode`): The formulae are displayed with all brackets/parentheses (value `full`) or with a minimum of brackets/parentheses according to the precedence and associativity of the operators (value `assoc`).
  - Output mode (`outputmode`): There are four output modes, `inputformat`, `ascii`, `latex` and `debug` which correspond to the output methods of `expr` (see Sect. 2.3.2).

  – Information level (`infolevel`): This variable controls the algorithmic information to be displayed (see also Sect. 2.3.3).

  The user can change these variables by the `set()` command or in the resources window of the XLWB . Within algorithms and output methods, they are only read.

- Global expressions (type `expr`):

  – Logical constants: Classical constants (e.g. `truesymbol`), linear logic constants (e.g. `botsymbol`).

  – Type symbols: Expressions that are returned when the *user* asks for the type of an expression (e.g. `listsymbol`).

  – An error variable (`errorsymbol`): It stores the last runtime error message of the interpreted LWB input language.

  Most of these global expressions are either logic independent or used by several logic modules. Therefore, they are entered in the user symboltable, e.g. `true`, `bot`, `LIST` (the type symbols are always in capital letters). They are protected against assignments.

- The type constants (integers): They are returned when logic module functions ask for the type of an expression (method `type()` of `expr`). For instance, `BOX` is the constant `102`, and to check whether the top-level operator of an `expr` instance `e` is a box, we can write:

```
if (e.type() == BOX) {
  ...
}
```

- Global output devices: See Sect. 2.3.1.

- Operator descriptions: The properties of the binary and unary operators of the LWB are combined in a record (class `opdescr` in exprrep.h). The descriptions are global to give a user interface the opportunity to display them and possibly even edit the non-constant parts (e.g. the sign for LaTeX output).

The global constants are defined in globals.h. The global variables are initialized in globals.C where also the operator descriptions are filled in. The global devices are in device.[hC].

## 2.3 The LWB Output

Typically, at the end of a statement (e.g. a function call or an assignment), the LWB interpreter sends the resulting expression to an output facility. There are three possible targets to where the output of the LWB (kernel and parser) can go:

1. To standard out (`stdout`) in the plain ASCII version.
2. To a file when saving expressions.
3. To the current output region of the XLWB.

All three cases must be supported by the same kernel and the same parser. A standard solution would be to split the program into a kernel process and an interface process which communicate with each other via pipes or sockets. However, this requires all data to be encoded by the kernel and to be decoded by the interface process, and vice versa when the user interface updates global variables that reside in the kernel. Such a design tends to lead to a 'shallow' user interface which is basically a remote control with little knowledge about the state of the system.

### 2.3.1 The Output Class `device`

The LWB output is passed to a global instance of a device class which is part of the kernel. There are different global devices to allow a classification of the information.

| | | |
|---|---|---|
| 1. | `dev` | device for normal output (especially expressions) |
| 2. | `warndev` | device for warnings and errors |
| 3. | `statdev` | device for status information |
| 4. | `infodev` | device for algorithmic information |
| 5. | `std` | 'standard' device |

Although a user interface can connect each of these devices to a different window, some devices may address the same real output channel. The ASCII version of the LWB , in fact, maps all to `stdout`.

Since the same kernel is used for different user interfaces, these global devices cannot be bound to a specific output facility. Therefore, they are declared as metadevices which contain only a reference to another device (see the class diagram 2.11). The user interfaces are supposed to assign a real device to each of these global metadevices. The class `metadevice` is a subclass of the abstract class `device` which provides the general interface. Devices can be used like C++ streams.[4] Furthermore, `device` offers the possibility to set and change indentations. The derived classes simply need to implement a `print()` method to display a given string, and a `flush()` method to clear the buffer (if the output facility maintains an own buffer). The class `osdevice` provides the connection to the C++ output streams (`ostream`) to which standard out (`cout`) and ordinary files belong.

[4]In C++, data is sent to an output stream (`ostream`) using the insertion operator `<<`. For example, the C++ style to write a string to standard out is `cout << '...';`.

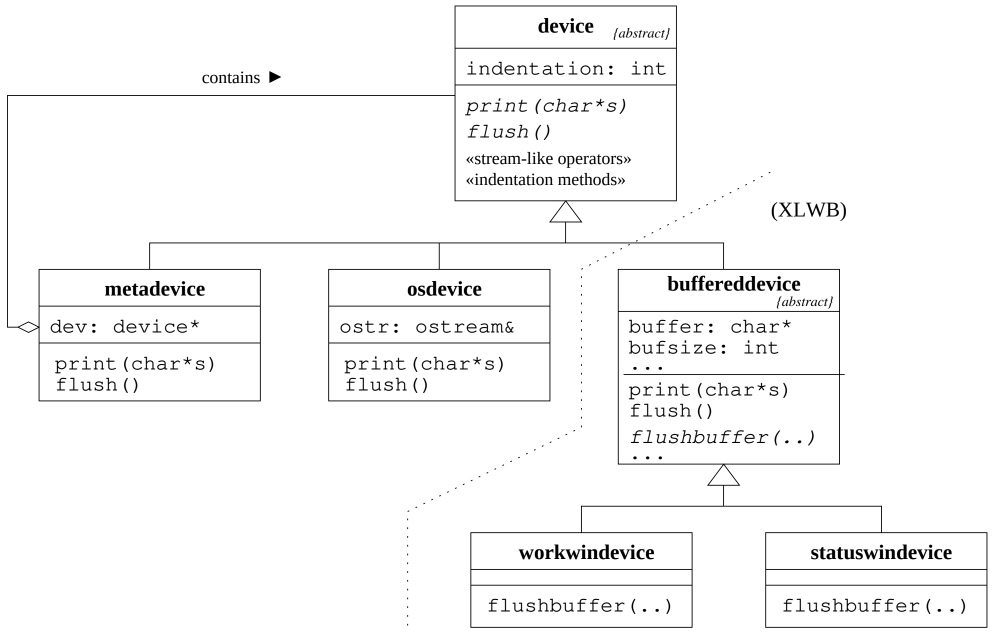



Figure 2.11: The device class hierarchy.

### Linking the Global Devices to the User Interfaces

The global meta-devices are only declared in the kernel (device.h) but not yet bound to a specific device type. The actual definition and the binding to a concrete device is done in interface specific files (plainversion.C for the standard LWB and parserconnection.C for the XLWB) in which they are initialized to different concrete devices. See Fig. 2.12.

## 2.3.2 Output Formats of Expressions

The LWB expressions can be displayed in four different output formats. The user can select the format for individual expressions using the commands `ascii()`, `inputformat()`, `latex()` and `debug()`, or for the whole session by setting the global variable `outputmode`:

```
> set("outputmode","inputformat");
        true
```

The methods to display an expression in different formats belong to the uniform set of operations which any subexpression must provide. As the other expression methods, the output methods of `expr` just call their counterpart in the referenced `exprrep`. Each subclass of `exprrep` knows how to print itself in the different formats. For instance, binary operators print themselves in infix notation taking the relative precedence into account to set or omit parentheses around operands. The device to which the output should go is the only argument of the output methods.

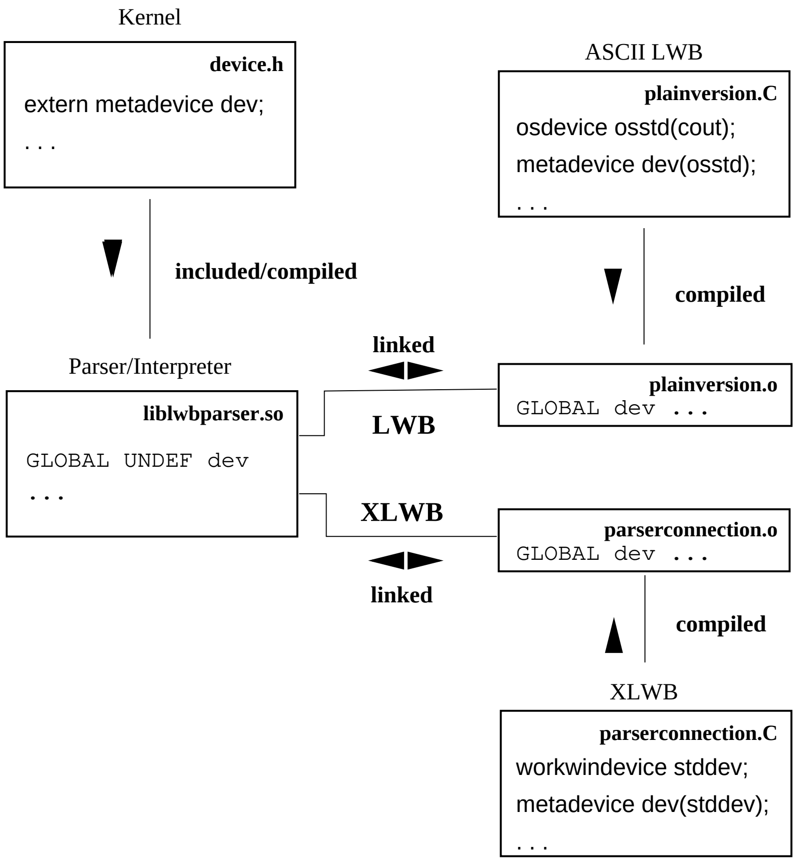



Figure 2.12: Linking the global metadevices to the user interfaces.

### The 'ASCII format'

In this format the logical operators are displayed in the best form using ASCII characters. In many cases, this format corresponds to the convenience input format and can be parsed again. See also Sect. 3.1.

### The Input Format

This method returns the expression in the unique input format which reconstructs the same expression when parsed again. This is useful for copy & paste or to save formulae of different logics in the same file. Note that not all expressions have a parsable input format, e.g. function descriptors and proofs cannot be parsed. For instance, the linear logic formula $a \otimes (b \oplus ?c) \multimap (a \wp b) \& c$ has the following input format.

```
ll> s:== a X (b + ?c) -o (a | b) & c;  # (linear logic mode)
ll> inputformat(s);
        a times (b llplus qm c) -o (a | b) with c
```

### The LaTeX Format

Each expression can be displayed in LaTeX code.

```
> latex( ~ box5 ~p  <-> dia5 p );
        \neg \Box_5 \neg p \leftrightarrow \Diamond_5 p
```

The output can be copied into a LaTeX document. When we typeset the LaTeX commands, we obtain $\neg\Box_5\neg p \leftrightarrow \Diamond_5 p$ .

### The Debug Format

Since debugging of linked structures is often tedious, there is a special debug format that includes internal data as for instance the reference count. The expression tree is displayed one line per node, and the lines are indented according to the depth of the node. The format of a single node is as follows:

```
operator (refcount), operand1, operand2, ..., internal data
```

The prefix notation is used to stress the top-level operator of the expression. For instance the debug format of the classical equivalence $(a \to b) \leftrightarrow ((\neg a) \vee b)$ is:

```
> debug( (a -> b) <-> (~a v b) );
<->(1), a -> b , (~a) v b
  ->(1), a , b
    a(3)
    b(3)
  v(1), ~a , b
    ~(1), a
      a(3)
    b(3)
```

Note that the reference counts (in parentheses) may differ from call to call, depending on the occurrence of `a` and `b` in other expressions.

### 2.3.3 Algorithmic Information

Many algorithms in logic, especially decision procedures, are very complex and may use a considerable amount of time for larger examples. Such algorithms have the possibility to inform the user about the state of execution. While for shorter examples the user may be interested in a full trace, for larger examples a rough description of the different stages is probably sufficient. To match the different preferences, the kernel proposes six levels of algorithmic information, ordered with respect to the supposed frequency of display. The user can then control the display of algorithmic information by setting the global variable `infolevel`:

```
> set("infolevel",3);
```

The number three corresponds to the kernel constant `ilev::where` in the table below.

| level | intended information | intended frequency |
|---|---|---|
| `ilev::none` | no information | ( never ) |
| `ilev::algo` | which algorithm | ( once, input independent ) |
| `ilev::subalgo` | which subalgorithm | ( $<$ 6 times, input independent ) |
| `ilev::where` | where are we, how much longer will it take | ( $<$ 20 times, as input independent as possible ) |
| `ilev::trace1` | detail level 1 | ( input dependent ) |
| `ilev::trace2` | detail level 2 | ( input dependent ) |

By choosing the appropriate level[5] the programmer classifies the information. The indentation facility of the `device` class is often used to indicate the current depth of recursion at which the information is generated. During the development, `info()` can also be used for 'scalable' debug output. However, the final versions should always restrict to information of interest to the user. See A.2.4 for the details how algorithmic information is generated.

As an example, we investigate the Peirce formula $((p \to a) \to p) \to p$ in intuitionistic logic. At the second detail level the intuitionistic prover issues the following algorithmic information (the different square brackets denote different groups of formulae within a sequent, e.g. literals, conjunctions, disjunctions etc.):

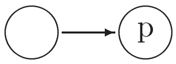
```
IPC |- A for given formula A ?
1.[],[],[],[]=>[],[],[((p -> a) -> p) -> p]
  (r ->)
1.1.[],[],[],[(p -> a) -> p]=>[p],[],[]
```

[5]The information levels (constants) of the left column are the first argument of the function `info()` (device.h).

```
    (l -> 4) with  (p -> a) -> p
1.1.1.1.[],[p],[],[]=>[p],[],[]
        (axiom).
1.1.1.2.[],[],[a -> p],[]=>[],[],[p -> a]
        (r ->)
1.1.1.2.1.[p],[],[a -> p],[]=>[a],[],[]
          (false).
        wrong  (.->.)->. -formula  (p -> a) -> p  selected
    |<--backtracking
    (false).

Given formula is not provable in IPC.
```

For a counter model we obviously need a world where $p$ is true and $a$ is false:



## 2.4 Functions as Expressions

The basis of a first order language are terms and relations where terms are usually built up by variables, constants and function symbols. Since a propositional language has no terms but just propositional variables (which can be viewed as 'null-ary relations'), there seems to be no need for functions as part of the language. Nevertheless, the LWB provides expressions of the form *name(args)*, where *name* is an expression (usually a symbol) and *args* is a comma-separated sequence of expressions. We will call such expressions in the following 'functional expressions', and *name* the 'functor'. Functional expressions serve two purposes:

1. More flexibility for the user to express formalizations of problems.
2. The access to the built-in functions of the logic modules and to the user-defined functions written in the LWB input language.

### 2.4.1 Higher Flexibility

It often occurs that subproblems of first order logic are essentially propositional, or vice versa, that propositional problems can be formulated more naturally in a first order style. For instance, the following formula `s` is propositionally valid (in classical logic) and is therefore accepted as input for the decision procedure.

```
cpc> s:== on(cup,table) v ~on(cup,table);
cpc> provable(s);
        true
```

When a functional expression is part of a formula, it is treated as a propositional variable. Of course, only syntactically equal functional expressions are identified; for instance, `on(cup,table)` is different from `on(table,cup)` or `on(cup,chair)`. Functional expressions can also be used to introduce user-defined n-ary operators, for example a triple disjunction.

```
cpc> s:= dis3(a, b, c) -> dis3(a & b, c, ~a);
        dis3(a, b, c) -> dis3(a & b, c, ~a)
```

The user has now two possibilities. First, he can write functions which interpret and transform formulae containing `dis3`. To do so, he can look at the zeroth operand to detect the new operator.

```
cpc> if s[1][0] = dis3 then
cpc>   print("Triple disjunction found!");
cpc> else
cpc>   print("Something else.");
       Triple disjunction found!
```

The second option is to write an expansion function `dis3` expressing the new operator in terms of old ones. Then all formulae containing the new operator will automatically expand when they are evaluated.

```
cpc> proc: dis3(x,y,z) begin return x v (y v z); end;
cpc> s;
        a v (b v c) -> a & b v (c v ~a)
```

For formulae defined before the definition of the expansion function, the expansion can be undone by unassigning `dis3`.

```
cpc> unassign(dis3);
cpc> s;
        dis3(a, b, c) -> dis3(a & b, c, ~a)
```

### 2.4.2 Connection to Built-In Functions

Functional expressions are the natural way to access the built-in logic module functions. Thus a call of a built-in function is considered as expression and can therefore be a part of another expression. Suppose that *f1(args1)* and *f2(args2)* represent two calls of module functions and *res1* and *res2* are the respective results. Then *f1(args1)* `->` *f2(args2)* is a valid LWB expression which is expected to evaluate to *res1* `->` *res2*. This implies, of course, that all functions at the input language level return again an expression. That is, they must be of the type $\texttt{expr}^n \rightarrow \texttt{expr}$ for some natural number $n$.

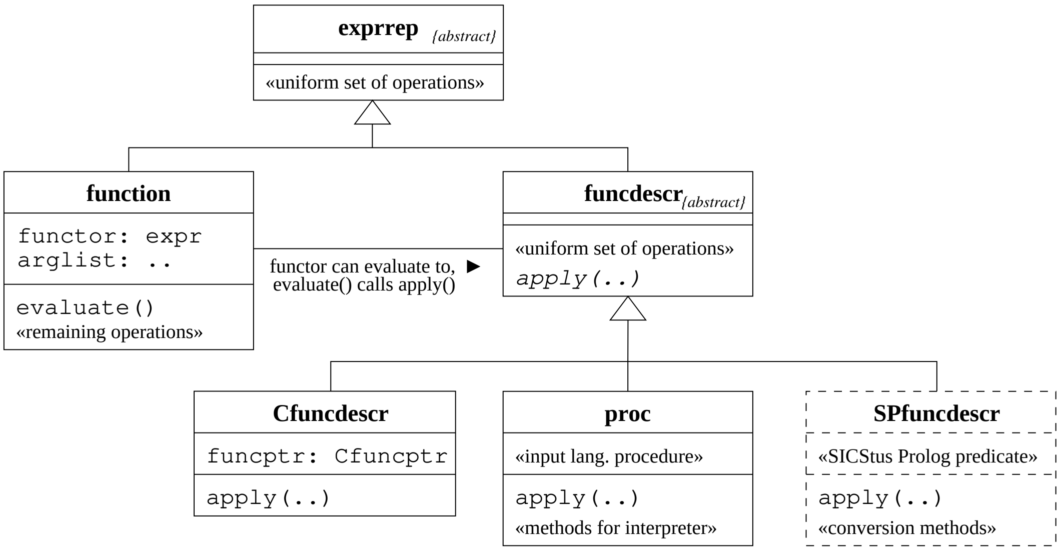



Figure 2.13: Function and function descriptor classes.

A functional expression (class `function`) consists of a functor and an argument list. The main difference to other expressions is the evaluation method. Provided that the functor evaluates to a function descriptor, the result of a real function call is returned. A function descriptor (abstract class `funcdescr`) is a special expression subclass to access executable functions (see Fig. 2.13). A derived class must simply implement an `apply()` method which is able to pass a list of expressions (arguments) to the internal function and to return the result as an expression. Figure 2.14 shows the call of a built-in function schematically.

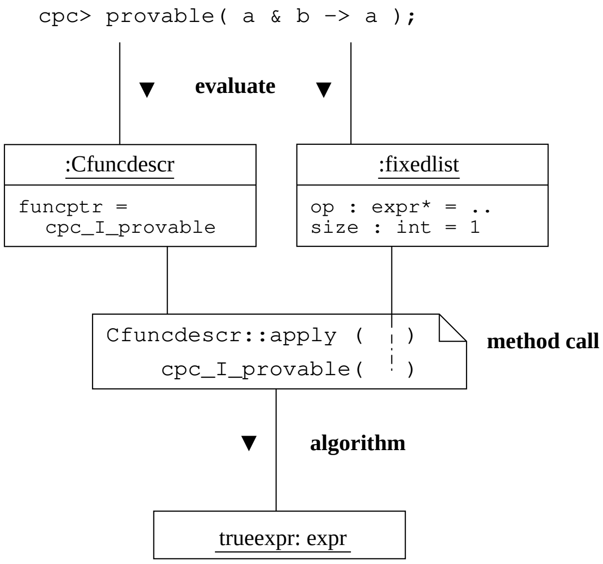



Figure 2.14: Call of a built-in function.

The class `Cfuncdescr` provides the access to the logic module functions by encapsulating a C function pointer. The C function accepts as argument the list of expressions that is passed to the apply method, and returns an expression. The class `proc` contains the description of a procedure written in the LWB input language. Its apply method induces the interpreter to continue with the represented input language procedure.

In order to access external algorithms written for other systems or in other languages we can add a new function descriptor. For example, functions written in SICStus Prolog can be accessed through the class `SPfuncdescr`[6] which encapsulates Prolog predicates. Typically, two conversion functions have to be provided, one to convert the arguments (expressions) to the foreign data type, and one to convert the result back to an expression. The interface to SICStus Prolog is exemplary for this type of connections to third party systems and is discussed in more detail in A.2.5 (p. 190).

[6]`SPfuncdescr` has been implemented and tested, but it is not included in the LWB 1.0.

## 2.5 Expression Structures

### 2.5.1 Lists and Arrays

Expression structures are mainly used for programming, both internally, in the logic module functions, and in user defined procedures written in the LWB input language. While internal functions may define their own data structures, the input language provides only arrays and lists. However, since these arrays and lists must again be expressions, we have the implicit flexibility to compose more complex structures. An inherent property of expressions is that they can be combined freely, and so we can construct lists of lists or a multi-dimensional array as an array of arrays. Lists and arrays can of course also be used by the logic module functions.

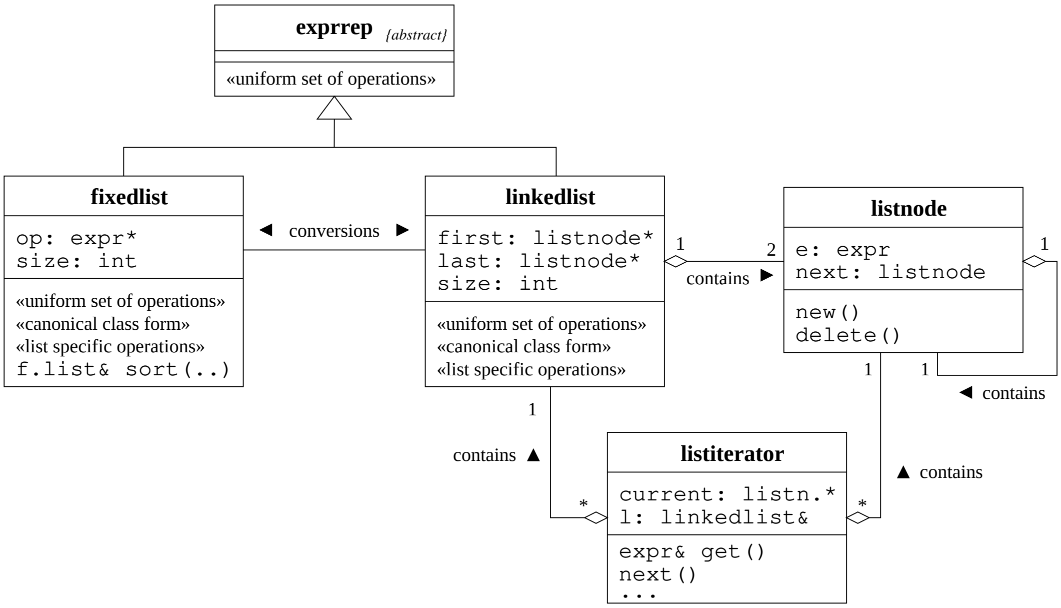



Figure 2.15: List classes.

As expression subclasses, the array class `fixedlist` (list of fixed size) and the list class `linkedlist` support the uniform set of operations. Moreover, both classes provide the same set of list operations (for this reason, the arrays are also called 'lists'). The left table of 2.1 shows the relative advantages of lists and arrays (a '+' means 'appropriate', and a '−' means 'possible but inefficient'). Depending on the most frequent usage, the user and the programmer can select the appropriate list representation, and if required, he can switch between lists and arrays using a conversion function.[7]

Note that a general expression has no access to the list specific operations because they do not belong to the uniform set of operations. They are only available after

[7]Both classes provide a constructor for the other class. In the input language the conversion functions are `convert2array()` and `convert2list()`.

| Frequent use for | list | array |
|---|---|---|
| random access | − | + |
| sequential access | + | + |
| size changes | + | − |

| List operations |
|---|
| Push & Pop of an element |
| Prepend of a list |
| Append of an element or a list |
| Concatenation (external function) |
| Member and emptiness check |

Table 2.1: Usage of lists and arrays, and list functions.

the expression is downcast to a list. The list operations of the input language take care of the necessary checks. Nevertheless, lists can directly be used as auxiliary structures, for instance by the logic module functions, that is, they need not play the role of an expression. Since instances of lists are used as any other C variable, both list classes are themselves in the orthodox canonical class form.

The list class `linkedlist` is the interface to a linked list of nodes each containing an expression. The node class `listnode` implements its own memory management by overloading the allocation and deallocation functions `new()` and `delete()` (the nodes are allocated in chunks). The list iterator (class `listiterator`) finally allows a sequential access of a linked list.

### 2.5.2 Expression Stacks

In the decision procedures of most logics, a specialized stack class is the key data structure. The algorithms usually proceed by applying rules of a calculus backwards until axioms (or possibly loops) are reached. Most calculi which are considered in this thesis enjoy the subformula property, which means that the formulae in the premise of a rule appear also as subformulae in the conclusion. So a backward application of a rule amounts basically to a formula decomposition by which the decomposed parts are usually classified according to their top-level operator (positive literals, negative literals, conjunctive formulae, disjunctive formulae etc.). When a rule has more than one premise, the algorithm is forced to replicate most of the formula structures which make up the conclusion. However, these structures can also be shared since they are only extended on deeper recursion levels. The LWB decision procedures employ a shared stack class to store collections of certain formulae. During a recursive execution, a 'tree' of shared stacks is built up which reflects exactly the branching behavior of the recursion. See, for example, Fig. 2.16. The numbers beside the nodes are reference counts.

While the use of shared stacks allows an optimal and efficient memory management, it restricts the operations to change a collection of formulae to push and pop. Especially, a change of the order, for instance a sorting operation, is not admissible.

The class `stack` is similar to `linkedlist` with a `stacknode` class analogous to `listnode`. There are two main differences: 1. The stacks are no expressions, that is, they are pure auxiliary data structures. 2. Each node is individually reference

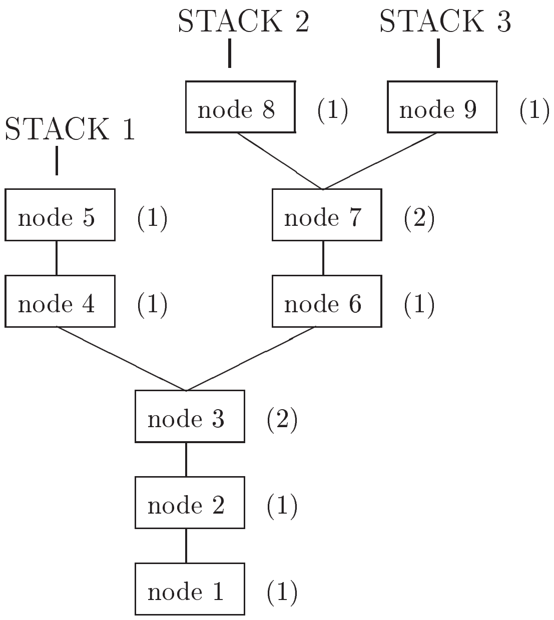



Figure 2.16: Memory sharing in the `stack` class.

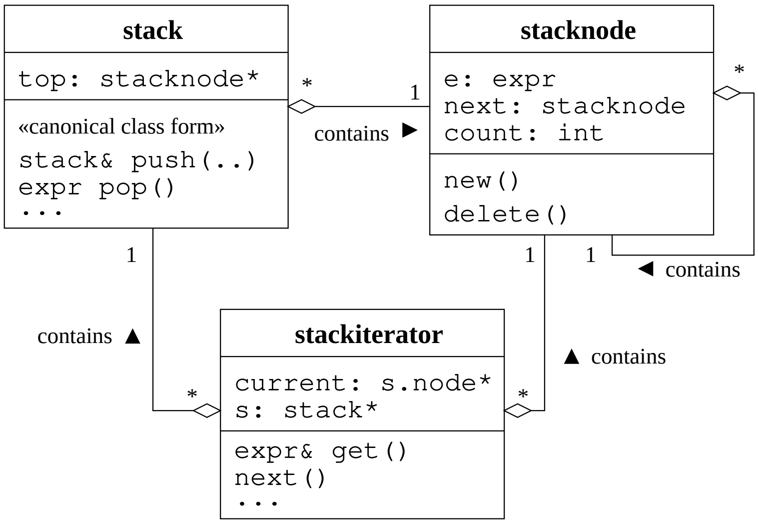



Figure 2.17: Stack classes.

counted, while a list, as a special expression, is only reference counted as a whole. The individual reference counting allows the sharing of stacks or, more generally, the sharing of common bottom parts among different stacks.

Furthermore, the stacknode has an additional integer field which is used to store the depth at which the node's formula has been decomposed. The use of this integer field is optional. There is also an iterator class `stackiterator` for sequential (read-only) access of stacks.

# Chapter 3

# A Language for Propositional Logics

In this chapter the design and implementation of a general language for computational (propositional) logic are discussed. The concept of the input language, as it is described in the following section, can be viewed as the 'external' design of the Logics Workbench: it is the input language through which the external world accesses the system. Since the language is the core of the interface to the user, it should meet the following postulates:

- User-friendliness: It should be easy to type in formulae of the various logics as well as commands.
- Comparability: It should be possible, in some way or another, to compare the different logics. This is a key goal of the LWB.
- Experimenting: The language must allow fast experimentation to qualify the system as a workbench.
- Automation: It must be possible to automate successful experiments, i.e. to extend the system with additional functionality.

The first three postulates are often fulfilled by graphical user interfaces (GUI), e.g. a formula editor for each logic might be perfectly user-friendly. On the one hand, if there is only a restricted language and only the GUI supports the features, the automation and extendibility become difficult if not impossible. On the other hand, having a powerful user language, selected tasks can still be supported by graphical means. For this reason, the language of the LWB is deliberately a part of the *user* interface and not just a (possibly hidden) interface between the kernel and some GUI. Consequently, the LWB is fully operational on a pure ASCII basis (the so-called ASCII LWB). Another invaluable advantage is that testing can also be automated. Since the entire functionality can be expressed in terms of the input language, test files written in this language can be composed and run in a batch mode. Conversely, GUI-oriented applications are often hard and tedious to test and automation requires a simulation of events, mouse-clicks etc. Automatic tests, as they are applied for the LWB 1.0, are the key to a stable system.

## 3.1 The Language Layout

In this section the design of the LWB language is described and it is shown that the above requirements are largely fulfilled. The full grammar in BNF syntax can be found in [1, p73] and a complete syntax description is available on the Web at

```
http://lwbwww.unibe.ch:8080/cgi-bin/LWBinfo?request=file&file=basic/basic_syntax.html
http://lwbwww.unibe.ch:8080/cgi-bin/LWBinfo?request=file&file=prog/prog_lang.html
```

After the introduction of the language, a longer example (sample session) is given where a simple verification problem is solved using temporal logic (Sect. 3.1.3).

The language can be characterized as follows:

- It is *untyped*.
- Anything is an LWB*expression*.
- There is a unique input format.
- The programming constructs are Pascal-like.

The language is untyped in the sense that the user is not required to declare the type of his symbols. The symbols can be used either as propositional variables in any logic or as metavariable for a formula in any logic or as metavariable for any other LWB data structure. The meaning of a symbol can also change during a session.

From the user's perspective anything he can type in is an LWB expression: variables, formulae of a particular logic, lists, arrays etc. These objects can be freely combined, reassigned and changed using a uniform set of operations. This flexibility is certainly convenient and renders declarations, re-declarations and cancelations superfluous. The check whether an expression is valid in a particular context (e.g. as argument to a logic module function) is deferred to runtime. However, such checks are in general inexpensive; they do not affect the efficiency since they are executed only once before an algorithm is carried out. Another advantage is the possibility to combine existing expressions into new 'data types': e.g. an expression of the kind '*list implies list*', which looks useless at first glance, might later be interpreted by a user-defined function as two-sided sequent.

Because of the wide range of incorporated logics it is impossible to have for all logical operators a 'natural' and unique representation in a standard character set. Both requirements, however, are essential: the representation must be natural and short to ease the use *within* a logic module, and it must be *unique* to allow comparability *across* logic modules. The solution of the LWB language is to provide two representations for each operator: 1. A convenience representation which is not unique, and 2. A unique representation which is (possibly) longer. The former, which is the default, is commonly referred to as 'operator overloading'. Consider, for example, the linear logic LL and the propositional linear time logic PLTL. In LL we have the binary 'times' operator ($\otimes$) and in PLTL the unary 'next-time' operator (X). In the LWB language both have `X` as the convenience format. The unique format, however, is `times` and `tX`, respectively. Note that in any other logic module `X` is simply a variable or metavariable.

Since no logic system can anticipate the needs of all users, a facility is necessary for programming new functionality so that individual users can expand the usefulness in directions they find necessary or interesting. The LWB input language comprises a powerful Pascal-like programming language offering all constructs needed to automate experiments and to implement new algorithms. The considerable efficiency of the interpreter along with the familiarity of the syntax qualifies the language for rapid prototyping. The average user can implement a small algorithm much faster since he can write his procedures in the flexible, untyped language he is already familiar with from using the LWB. Moreover, the interpreter allows a step-by-step testing of statements which is more efficient than the usual test-compile-link cycle. Auxiliary data structures needed for programming, such as lists and arrays, are directly supported including the corresponding manipulation functions. The closeness to Pascal is for most users an advantage; the absence of system specific features makes it very easy to learn and apply. We also refer to symbolic algebra tools (e.g. Maple [39, 13]) which offer similar procedural languages.

### 3.1.1 The Basic Language

In the following section, we focus on the syntax and thus omit the output when it is basically the same as the input.

We use `a`, `b`, `e1`, `e2`, `e3` in the examples below as variables. If they were metavariables, the input would still be valid; in the output, however, the represented expressions would be displayed instead. The '`>`' at the beginning of an input line is the prompt. Everything on the input lines to the right of the delimiter '`#`' is treated as comment and ignored by the interpreter. Statements must be terminated by a semicolon. If several statements are entered at once, the output contains at least one line for each.

#### Simple Expressions

There is a small set of simple or primitive expressions such as symbols, strings, and integers. While there are no restrictions for strings, symbols must start with a character and may not contain special characters (except possibly underscores).

Strings must be enclosed in double quotation marks.

```
> my_symbol;
> "my string";
> 123;
```

### Logic Dependent Modes

Currently, there are three language modes for three groups of logics. The modes differ in that the convenience representation of some logical operators is only available if the corresponding mode is active.

1. The default mode for the largest group containing classical, intuitionistic, modal, and non-monotonic logics.
2. Temporal mode for temporal logics. Here, some capital letters have a meaning as operators.
3. Linear logic mode. Especially, the plus sign '`+`' is overloaded (usually the integer addition which has the unique representation `plus`) as well as the question mark '`?`' (usually an equivalent for `help`).

### Composed Expressions

Among the composed expressions the logical operators play the most prominent role. They are summarized in Table 3.1 and 3.2. The latter table lists the operators in the order of their precedence. In the 'Default' column of Table 3.1 the syntax for the logic dependent convenience format is given, and in the 'Unique' column the one for the unique input format.[1] For some operators a verbatim alternative without special characters is available; the alternatives are listed in the 'Alt' column. Finally, the 'LaTeX' column shows the commonly used graphical symbols (this is only an output format).
For instance, the multi-modal formula $\Box_1(a \vee b) \wedge \Box_2(a \vee b) \wedge \neg a \rightarrow \Box_2 \neg a$ can either be typed in as

```
> box1 (a v b) & box2 (a v b) & ~a -> box2 ~a;
```

or equivalently as

```
> box1 (a or b) and box2 (a or b) and not a imp box2 not a;
```

A special type of composed expressions are the functional expressions.

```
> f(e1);  g(e1, e2);  h(e1, e2, e3);
```

[1] In the help pages the default format is also called the 'ASCII' format since it is the closest approximation in the ASCII character set, and the unique input format is simply called 'input format'.

| Logic(s) | Operator | Default | Unique | Alt | LaTeX |
|---|---|---|---|---|---|
| classical | conjunction | `&` | `and` | `and` | $\wedge$ |
| | disjunction | `v` | `v` | `or` | $\vee$ |
| | implication | `->` | `->` | `imp` | $\rightarrow$ |
| | equivalence | `<->` | `<->` | `eq` | $\leftrightarrow$ |
| | negation | `~` | `~` | `not` | $\neg$ |
| modal | necessity | `box` | `box` | | $\Box$ |
| | possibility | `dia` | `dia` | | $\Diamond$ |
| multi-modal | necessity | `box3` | `box3` | | $\Box_3$ |
| | possibility | `dia5` | `dia5` | | $\Diamond_5$ |
| temporal | always (past) | `H` | `tH` | | $\mathsf{H}$ |
| | sometimes (past) | `P` | `tP` | | $\mathsf{P}$ |
| | always (future) | `G` | `tG` | | $\mathsf{G}$ |
| | sometimes (future) | `F` | `tF` | | $\mathsf{F}$ |
| | next-time | `X` | `tX` | | $\mathsf{X}$ |
| | until | `U` | `tU` | | $\mathsf{U}$ |
| | before | `B` | `tB` | | $\mathsf{B}$ |
| | all futures | `A` | `tA` | | $\mathsf{A}$ |
| | some future | `E` | `tE` | | $\mathsf{E}$ |
| linear | storage, bang | `!` | `!` | | $!$ |
| | question mark | `?` | `qm` | | $?$ |
| | times | `X` | `times` | | $\otimes$ |
| | par | `\|` | `\|` | `par` | $\wp$ |
| | with | `&` | `with` | `with` | $\&$ |
| | plus | `+` | `llplus` | `llplus` | $\oplus$ |
| | implication | `-o` | `-o` | `llimp` | $\multimap$ |
| | equivalence | `o-o` | `o-o` | `lleq` | $\multimapboth$ |

Table 3.1: Logical Operators.

The main purpose of functional expressions is of course to serve as an 'interface' to the built-in and the user-defined functions (e.g. `provable(..)` for the decision procedures). However, if neither of `f`, `g`, `h`, `e1`, `e2`, `e3` has a special meaning, i.e. is used as metavariable, the LWB simply returns the input, that is, the functional expression stands for itself. This offers additional flexibility. By choosing an unused symbol as functor, new operators can be defined. For example, $\not\wedge$ (not and, 'nand') could be mimicked by the prefix notation `nand(.,.)` which could then be interpreted by user functions. In the same way first order constructs could be introduced: for instance

```
forall(x, P(x) & exists(y, Q(x,y)))
```

is a valid LWB expression.

Lists and arrays are composed expressions that store groups of other expressions. Lists are delimited by brackets; the empty list is denoted by `[]`. For arrays the keyword `array` is used.

```
> []; [e1, e2];
> array(); array(e1,e2);
```

Note that the entries of list or arrays may again be lists or arrays.

### Operations on Expressions

There is a uniform set of operations which are applicable to *all* expressions. The most important ones are operand extraction (indexing), query of the number of operands, query of the operator type, and substitution.

```
> (a & b)[1];  (a & b)[2];
        a
        b
> nops(a & b);  type(a & b);
        2
        AND
> (a & b) { e1/a, e2/b };
        e1 & e2
```

Indexing, `nops()` and `type()` always refer to the topmost operator. The substitution can be entered in a textbook-style manner using curly brackets. Only symbols, not subformulae, can be substituted (for efficiency reasons).

### Statements

Beside the display of expressions, which includes the display of function call results, the assignment is the most important statement. The assignment operator is `:=`, as in many other programming languages (`=` is the equality operator). There is also a silent assignment, an unassigment, a protection and an unprotection statement. The silent assignment suppresses the output. This is useful if we want to store a possibly long result of a function call without actually displaying it. It is also often used to save expressions in a file. By the protection statement symbols can be protected from reassignment.

```
> a := e1 & e2;
        e1 & e2
> a :== e1 & e2;  # silent assignment
> protect( a );  a:= e1 -> e2;
ERROR in assignment: protected symbol
> unprotect( a );  a:= e1 -> e2;
        e1 -> e2
```

The silent assignment, as well as `protect()` and `unprotect()` produce no output.

### Evaluation

By an assignment, a symbol can turn from a propositional variable to a metavariable. If the symbol is part of previously defined formulae, the confusing situation might occur that at the same time in a session the symbol is propositional variable *and* metavariable. The solution is to treat the symbol also in previously defined formulae as a metavariable. Therefore, whenever the value of such a formula is asked, an 'evaluation' must take place which substitutes the metavariables by their values. The evaluation rules are quite simple:

1. Any expression on the right hand side of a statement is evaluated down to symbols which are not assigned. The resulting expression is used (e.g. for assignment).

2. In function calls the arguments are evaluated first, i.e. no (re)evaluation is ever needed in function bodies.

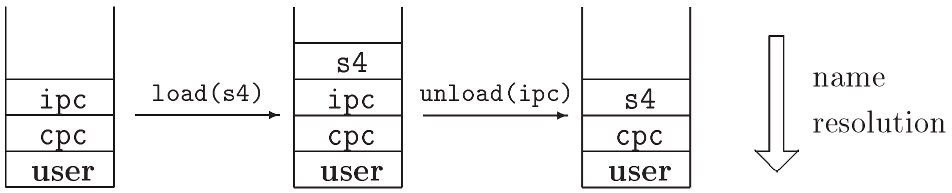
Evaluation is the price we have to pay for the convenience of an untyped language. See also Sect. 2.2.1.

```
> s:== a & b;   # silent assignment
> s -> a;       # s need evaluation
        a & b -> a
> b:== e1 v e2; # silent assignment
> s -> a;       # now also 'b' within s needs evaluation
        a & (e1 v e2) -> a
```

**Module Concept**

Logic modules can be loaded as well as unloaded. Several modules may be loaded at the same time. Since related functions of different modules have deliberately the same name, e.g. `satisfiable()`, a conflict resolution must take place when a user enters just the common name. The loaded modules are organized as a stack, that is, the top module is the last loaded. The bottom module, it is called the *user module*, plays a special role. It is not a logic module but serves as a data base where general constants such as `true` and `false` and the user defined symbols are stored. The name resolution using the module stack is similar to the command resolution in UNIX when the `PATH` variable is defined. The rules are:

1. If a symbol does not occur in a loaded module, it is entered into the bottom module (the user module).
2. Otherwise it is taken from the first module where it occurs in the top-down traversed stack.




A symbol (usually a function name) in a specific loaded module can be addressed using the module selection operator '`::`'. In this way, functions and algorithms of different logics can directly be compared.

```
> load(cpc,ipc);            # classical and intuitionistic logic
        ipc  cpc  user
> cpc::provable( a v ~a );  # provable in module 'cpc'
        true
> provable( a v ~a );       # topmost provable  (intuitionistic)
        false
```

See also Sect. 3.5.

## 3.1.2 Programming Language Constructs

In the first place the programming language has been designed to facilitate larger experiments by partial or full automation. Another purpose is to enable personal adaptions: for instance, if a user wants to call some function always with the same option, he can write a simple wrapper procedure which does the job. Note that if a long but frequently used function name should only be abbreviated, a plain assignment is sufficient:

```
>  cpr:= cpc::provable;
        BUILTIN
```

The definition of a new operator in terms of existing ones is equally simple. For instance, if a negated equivalence is often used, we can define the following little procedure:

```
proc neg_eq(x, y) begin
  return not(x <-> y);
end
```

With little effort, a user can customize the LWB according to his needs and thereby reduce tedious, repetitive typing.

However, we should point out that the efficiency of the internal organization along with the comprehensive programming constructs give sufficient support to also implement larger algorithms such as simple provers.

### Compound Statements

The following if–then–else statement and the various forms of loops can be used within procedures as well as interactively.

```
> if x=b then
    list := [];
  else
    list := [x];

> foreach i in [a, b, c] do
    print(i);

> for i := 1 to nops(s) do
    s[i] :== not s[i];

> while x > 3 do
    dec(x);
```

### List/Array Operations

The most common list operations are provided allowing easy handling of collections of formulae. For convenience, the same set of operations equally works on arrays. However, if the operation changes the size of the array, the corresponding list operation is faster. On the other hand, random access is more efficient on arrays.

```
> list:== [e1,e2]; list2:== [e3];  # (silent assignments)
> push(a,list);
        [a, e1, e2]
```

```
> append(list,b);
        [a, e1, e2, b]
> pop(list);
        a
> list[2..3];    # sublist extraction
        [e2, b]
> concat(list,list2);
        [e1, e2, b, e3]
```

While `push()`, `append()` and `pop()` are in-place operations changing `list` itself, the sublist extraction and `concat()` are 'functional', that is, they return a new list leaving the arguments unchanged. The function `pop()` returns the first element which is popped off the list.

### Procedure Definition

The procedure definition resembles Pascal except that there are no types. The main features are:

- Call by value and call by reference: A reference argument is preceded by the `var` keyword.
- Local variables: It is possible to have local variables using the `local` keyword. They may be named as existing global variables or variables of an embracing scope. Local variables are always initialized with the empty NULL expression.
- Local procedures: Inside procedures, before the `begin` keyword, local procedures can be declared. Thus, procedures can be nested and encapsulation is possible.
- Lexical scoping (static).
- Re-entrance: Direct and indirect recursion are permitted.

```
> proc: f( x , var z)
    local y;
    proc f1() begin
      return y & x;       # local y, parameter x
    end;
  begin
    y:= e1;               # local y, global e1
    z:= e2;               # parameter z, global e2
    return f1() ;
  end;
> f( a, b );
        e1 & a
> b;
        e2
```

### Error Handling

Errors, such as trying to pop from an expression other than a list, result in a runtime error which causes the immediate exit from all active procedures and the return to the user prompt where an error message is displayed.

```
> a:== b & c;
> proc: foo() begin pop(a); end;
> foo();
ERROR in pop: list or array expected
(called by foo)
```

Occasionally, the programmer may want to raise errors explicitly, e.g. if an argument is not of a certain form.

```
if type(x) <> LIST then raiseerror("List expected");
```

It is possible to catch errors within procedures and to branch on whether an error has occurred and possibly on the type of the error. This is helpful if the called function is 'unsafe' (for instance, if the programmer is not sure about the arguments which are accepted by the called function). The current error message is stored in the global `error` variable.

```
r := catcherror(call_unsafe());
if r = error then
  print("an error occurred within call_unsafe()");
```

### Debugging

A limited debugger can be entered by typing

```
> debugger function( args )
```

The debugging mode provides a small and exclusive set of commands for basic debugging. The commands in parentheses are rather for internal debugging.

| command | purpose |
|---|---|
| `q;` | exit debug mode. |
| `s;` | step: execute code (primitive instructions) until the end of the next statement is reached. |
| `n;` | next: as step but does not enter other functions. |
| `p x;` | print identifier `x`. |
| `w;` | display the stack of alive activation records with values of the local variables. |
| ( `si;` ) | step instruction: execute one primitive operation. |
| ( `d x;` ) | print identifier `x` in debug format. |
| ( `ps;` ) | print temporaries of the top activation record. |

### 3.1.3 A Sample Session

We present an example which shows how a non-classical logic can be treated in the LWB. The problem to be solved is illustrated in Fig. 3.1.

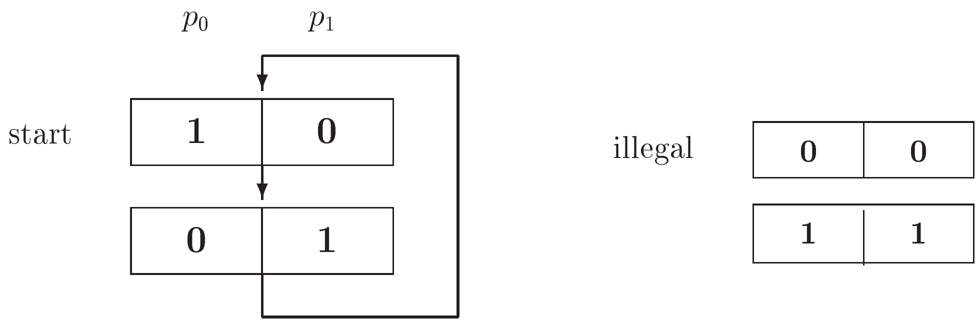



Figure 3.1: A 'flip'-machine.

We have a 2-bit 'machine' in which the first bit is initially 1 and the second 0. Once started, the machine keeps flipping the bits from 1 to 0 and from 0 to 1, respectively. Written as a sequence, we have the following values:

$$(1,0) \to (0,1) \to (1,0) \to (0,1) \to \cdots$$

The task is to find an algorithm which exhibits such a behavior and to prove that it is correct. We will formalize an algorithm as a formula in the propositional linear time logic PLTL. We will come back to PLTL in Chap. 6 where the logic is properly introduced (syntax, semantics) and where a new tableau calculus is presented. Here we give only an informal description of the employed non-classical operators:

| Temporal operators | Informal semantics |
|---|---|
| $\mathsf{X}\,p$ | next time $p$ |
| $\mathsf{G}\,p$ | always $p$, $p$ is going to be |
| $\mathsf{F}\,p$ | sometime $p$, there is a future where $p$ |

A possible formalization of an algorithm is the following formula:

$$\text{flip} \;:=\; \mathsf{G}(\,(\mathsf{X}p_0 \leftrightarrow \neg p_0) \land\; (\mathsf{X}p_1 \leftrightarrow \neg p_1)\,).$$

It simply says that the value of the bits in the next state ($\mathsf{X}\ \ldots$) is always ($\mathsf{G}(\ldots)$) the negation of their current value (we interpret negative literals as 0 and positive literals as 1). Accordingly, the start state, the following state, and the two 'illegal' states (which should never occur) can be written as:

$$\begin{aligned}
\text{start} &:= p_0 \land \neg p_1 \\
\text{follow} &:= \neg p_0 \land p_1 \\
\text{illegal}_1 &:= p_0 \land p_1 \\
\text{illegal}_2 &:= \neg p_0 \land \neg p_1
\end{aligned}$$

First, we load the PLTL module. The special mode for temporal logic is entered in which the capital letters G, F, and X can be used as operators.

```
> # ______________________________________________
  #
  #    A simple 2-bit flip-machine
  # ______________________________________________

  load(pltl);
             pltl  user
Entering special mode for tense & temporal logics.
```

Figure 3.2: Load the PLTL module.

Then we define the specification of the algorithm, the start state, and the illegal states. We use silent assignments.

```
> start :== p0 & ~p1;              # (1,0)
  follow:== ~p0 & p1;              # (0,1)
  illegal:== [p0 & p1, ~p0 & ~p1]; # (1,1) and (0,0)

  flip:== G ( (X p0 <-> ~p0) &     # .. (1,0)->(0,1)->(1,0)-> ..
              (X p1 <-> ~p1) );
```

Figure 3.3: Definition of the states.

Now we can check some properties: two liveness properties, a safety property and the equivalence to an alternative definition.

```
> provable( flip & start -> G F start );      # liveness prop.
  provable( flip & start -> G F follow );

  nothing_bad:== ~illegal[1] & ~illegal[2];
  provable( flip & start -> G nothing_bad ); # safety prop.

  alt_flip:== G( (start -> X follow) & (follow -> X start) );
  provable( flip & start  <->  alt_flip & start );
             true
             true
             true
             true
```

Figure 3.4: Verification of properties.

However, there is a problem: we cannot guarantee that the algorithm alone will eventually lead to the start state. So we compute a counter-model. A model is given as a list of states with additional numbers indicating the respective successor state. The counter-model shows that 'flip' loops between the illegal states.

```
> provable( flip -> F start );
  countermodel( flip  -> F start );
             false
             [[[~p1, ~p0], [2]], [[p0, p1], [1]]]
```

Figure 3.5: A counter-model.

Finally, we define a 'self-correcting' algorithm which satisfies the guarantee property and the previous liveness properties, regardless of whether we start in an illegal state.

```
> flip2:==  G ( (X p0 <-> ~p0 & p1) &
                 (X p1 <-> ~p1) );

  provable( flip2 -> F start );
  provable( flip2 & (illegal[1] v illegal[2]) ->
                G F start & G F follow );
             true
             true
```

Figure 3.6: A 'self-correcting' algorithm.

## 3.2 Lexical Analysis

Now we turn to the 'internal design' of the LWB input language. We present a description of the components which are most directly connected with the input language: the lexical analyzer, the actual parser, the interpreter, the implementation of the module concept. These components have strong interrelations and are therefore combined in a single library called liblwbparser.so. Figure 3.7 presents an overview and shows the relations to the other LWB components (it can be viewed as a refinement of Fig. 1.1).

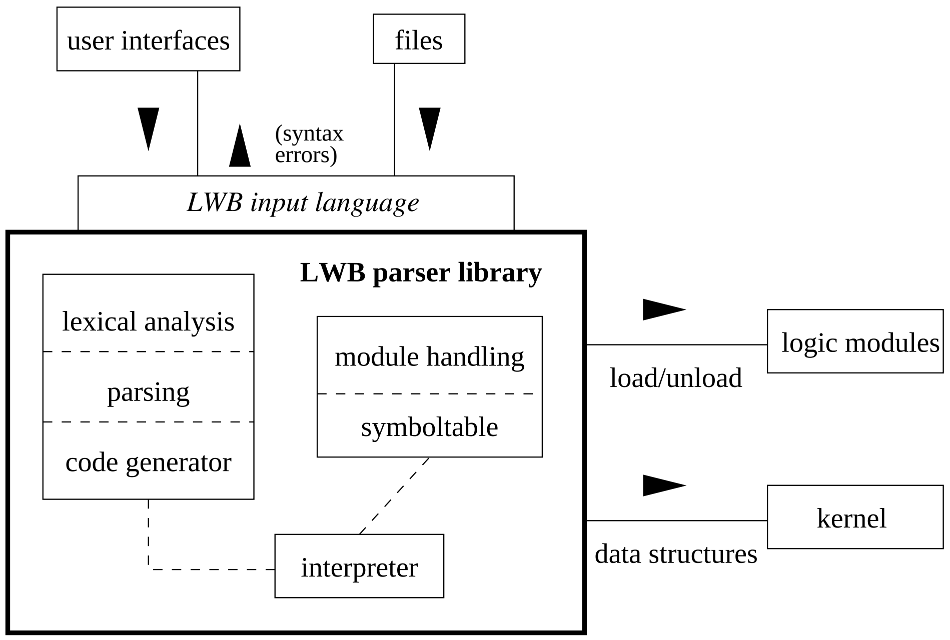
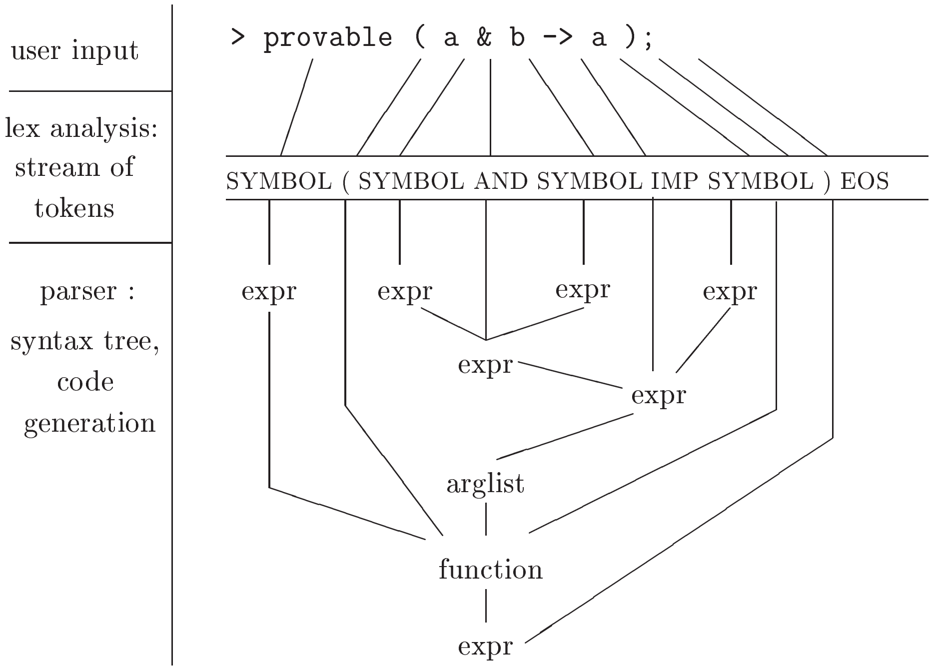



Figure 3.7: The LWB parser library.

Before we discuss the lexical analysis, we present a scenario to give an idea what the tasks of the various elements are.




For the above example the following meta-code is generated:

```
L263004 global  provable
L263016 global  a
L263028 global  b
L263040 and
L263052 global  a
L263064 imp
L263076 invoke  1
```

### Tasks of the Lexical Analyzer

Of course, the chief purpose of the lexical analyzer is the token recognition. Tokens are usually defined in terms of regular expressions. Since the lexical analysis is the first component in the chain of input processing, it has to deal with files and the associated problems.

- Reading either from files or from strings and switching between both: Reading from strings is used when the parser is connected to the GUI.
- Nested reading from files: It occurs when a LWB-statement `read(next.lwb)` is found within a parsed file.
- Switching between default state and special states: A special state is for instance when the LWB is in debugging mode in which additional commands (tokens) are recognized, or when a logic is entered requiring many non-standard operators (e.g. linear logic).

- Parse error localization: The input must be buffered to redisplay an erroneous line.

The lexical analyzer for the LWB is produced with lex, a tool for generating scanners. lex generates as output a C source file, which defines a routine `yylex()`. This routine analyzes the input for occurrences of regular expressions. Whenever it finds one, it executes the corresponding C code, which usually contains a `return` statement passing back a value from `yylex()` to its caller, the parser generated by yacc. The value passed back is essentially a token constant. The next time `yylex()` is called, the scanner picks up where it left off. For the LWB flex, the GNU-version of lex, has been used, which offers a lot more flexibility for various subtasks. See Fig. 3.8 for the relation of the lexical analysis and the parser.

### 3.2.1 Format of the Input File

The lex input file consists of three sections, separated by a line with just `%%` in it:

*definitions*
`%%`
*rules*
`%%`
*user code*

In the definitions and rules sections, any indented text or text enclosed in `%{` and `%}` is copied verbatim to the output (with `%{` and `%}` removed). The `%{` and `%}` must appear unindented on lines by themselves. `%{}`– text appearing before the first rule is used to declare variables that are local to the scanning routine.

#### The Definitions Section

The definitions section contains declarations of simple name definitions to simplify the scanner specification. Name definitions have the form:

*name definition*

The definition can subsequently be referred to using '{*name*}' which will expand to *definition*. For example,

```
and     "&"|"and"
not     "~"|"not"
```

defines {`and`} to be a regular expression which matches a single ampersand or the string `and`, and {`not`} to be a regular expression which matches a single tilde or the string `not`.

### The Rules Section

The rules section of the lex input contains a series of rules of the form:

*pattern action*

where the pattern must be unindented and the action must begin on the same line. The above defined names can be used as patterns (or parts of patterns):

```
{and}           { tokenpos += yyleng; return tAND; }
{not}           { tokenpos += yyleng; return tNOT; }
```

In this case a local variable is updated and a token is returned (tokens are defined by a parser output-file from `yacc`). A typical part of the actions is to update the token position (`tokenpos`) by the current token length (`yylen`, a lex variable) in order to identify the erroneous part within the current input line in case of a parse error (see also error localization on p.63). Note that the ordering of the rules matters! Only the action of the first matching pattern is executed. The last rule is typically of the form

```
.               { tokenpos += yyleng; return yytext[0]; }
```

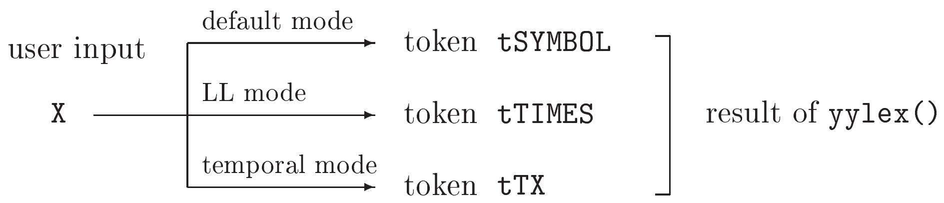
catching 'anything else' ('`.`' matches any character). If returned to the parser, as it is done here, it will most certainly cause a parse error.

### The User Code Section

Finally, the user code section is simply copied to the C file verbatim. It is used for companion routines which call or are called by the scanner.

## 3.2.2 Special Modes

There are currently four modes in which the lexical analysis can be put. There are three 'logical modes' (cf p. 46) and one mode for the debugging (cf p. 53). Depending on the mode, the lexical analyzer returns different tokens. Consider again the capital letter `X`. It can have three different meanings:




The special modes, also called start states or start conditions, are declared in the definitions section. There are two forms of declarations:

`%s` *mode-name* (non-exclusive mode)
`%x` *mode-name* (exclusive mode, only flex)

In the rules section, certain rules can now be labeled with <*mode-name*> so that they are only considered when the mode is active.

<*mode-name*>*pattern action*

In an exclusive special mode (declared by `%x`), only labeled rules can trigger, while in a non-exclusive mode the labeled rules can trigger additionally. For the LWB we have three mode declarations (the default mode need not be declared).

```
/* special state for linear logic and tense logic */
%s LL
%s TK
/* exclusive special state for debugging a procedure */
%x sDEBUG
```

The debugging mode is exclusive, because only a restricted set of commands should be used when the debugger is active. Note that if a rule of a non-exclusive mode is to supersede an unlabeled default rule, it must be listed before:

```
<LL>&             { tokenpos += yyleng; return tWITH;  }
{and}             { tokenpos += yyleng; return tAND;   }
```

Special modes are entered by the C statements '`BEGIN` *mode-name*`;`', and left by '`BEGIN INITIAL;`'. The latter just returns to the initial or default mode.

### 3.2.3 Input to the Lexical Analyzer

As a front-end of the parser, the lexical analyzer has to cope with input from different sources.

1. Input from standard input (stdin).
2. Input from strings.
3. Input from files.

The original input source is either a string or the standard input. It never happens in the LWB that the input source switches from 1. to 2. or vice versa. However, if the user input or the input string contains a `read(` *filename* `)` statement, the input source is switched to the file *filename*, and after the file is processed, it switches back. If a `read()` statement occurs also in *filename*, the input source is switched from one file to the other, and so on.

### Input from Stdin

Reading from stdin and writing to stdout is the default of the ASCII LWB. It is also the basis for I/O redirection as it is common for UNIX applications:

```
lwb < test.lwb > test.lwbout
```

This enables the LWB to be used in batch mode, which is, as suggested by the example, heavily employed for automatic tests. Another application of this kind is the usage of the LWB via the World Wide Web (WWW), where the LWB is fed by a Perl program which in turn gets the input from a HTTP (Hypertext Transfer Protocol) server.

### Input from Strings

The XLWB communicates with the parser through strings. When the user evaluates an input region, the function

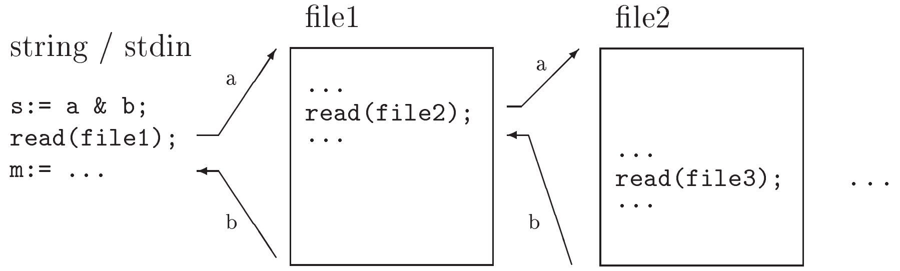
```
int yyparse(char* s)
```

is called with the input region string as argument. This is an overloading of the standard yacc-generated function `int yyparse()` which takes no argument.

Note that an input region and thus the string to be parsed may contain several lines with LWB statements. This must be taken into account when parse errors are localized. For a more detailed discussion of string input we refer to the appendix A.3.1.

### Input from Files

When a user deals with large pieces of LWB input, for instance, data bases of formulae or procedure definitions, he will commonly use a separate editor, especially in connection with the ASCII LWB. After the editing, he reads the file into his LWB session. In the same way a session status is reloaded which has been stored to a file using the `save()` command. To allow splitting and modularization of projects, nested reading of files is supported, i.e. files may themselves contain `read()` statements.




a: lex function `readfromfile()`

b: lex function `yywrap()`

Internally, the function `void readfromfile(const char *name)` is called to put the lexer in the appropriate state when a new file is entered, and `int yywrap()` is called when the end of the file is reached. The switching between files relies heavily on flex specialities, which facilitate the job a lot. The scanning itself is identical to input from stdin. For the nested parsing of files, a stack of states is required, in which the current state of the lexer can be stored. This stack of parser states is a typical representative of an auxiliary data structure defined in the user section. A technical explanation of the file-switching can be found in A.3.1.

### Parse Error Localization

We have tried to provide a precise and reliable error localization for all variants of the parser usage, including complex cases such as nested file reading. Although it complicates the reading of the different sources (see A.3.1 for details), it is worth the effort since precise error messages are an important aid for the user. The main difference when the source is stdin is that the chunk of analyzed data always corresponds to just one input line, while in the other modes there are usually several input lines.

Assume, for example, that the 11th line of the file errortest.lwb is

```
a and and b;
```

Then the LWB reports the following error message when it executes the statement `read(errortest.lwb)`:

```
errortest.lwb,line 11: parse error at 'and':
a and and b;
      ^
```

In the XLWB the (first) erroneous token is highlighted and can directly be typed over (see Fig. 4.5 on page 99).

## 3.3 The Parser

The parser recognizes the accepted LWB language in terms of lexical tokens, and in parallel, the code generator produces the corresponding meta-code. In interactive mode, the LWB interpreter executes the generated meta-code whenever a statement has been parsed successfully. In this section, we concentrate on the more general aspects of parsing a language for propositional logics and discuss briefly why the seemingly natural Prolog approach was rejected. Of course, since the parser has to deal with a comprehensive language, it is complex itself, and it is not possible to address all aspects here. Especially, specific problems related to the meta-code generation for the LWB programming language are deferred to Sect. 3.4.3 (after the introduction of the LWB interpreter). We focus on the following topics:

1. Relation to the lexical analysis.
2. Formulation of parser rules for logical expressions.
3. The basic treatment of intermediate results in the bottom-up recognition process.
4. Error handling.
5. Specialities and problems related to the yacc solution.

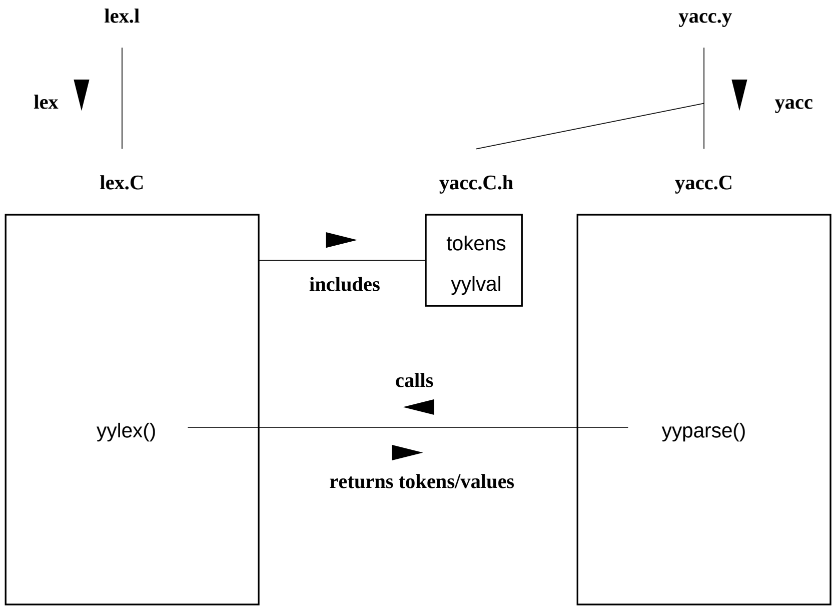



Figure 3.8: LWB parser: relationship lex – yacc.

The LWB parser is produced with the parser generator yacc. Yacc takes a context-free grammar as input and writes an LALR(1) (cf. Sect. 3.3.5) parsing algorithm in a C file. The entry point to the parser is `yyparse()`, which returns zero if the parse succeeds and non-zero if not. As opposed to `yylex()`, each call to `yyparse()` starts parsing anew, forgetting whatever state it might have been in the last time it returned (see also the graphical overview in Fig. 3.8). For the LWB , bison, the GNU-version of yacc, is used.

### 3.3.1 General Discussion

#### Prolog Parsing

Many people would argue that writing a Prolog parser is much simpler and more appropriate in a logical context. In fact, the parsing of expressions almost comes for free in Prolog. However, with regard to more complex structures such as programming language constructs, a Prolog solution gets quickly difficult and often inefficient. Prolog is not really suited for the base language of a complex system. There are no widespread tools, and the interfaces to the operating system, to other programming languages or to graphical user interfaces do either not exist or differ from dialect to dialect. This limits in general the portability to different platforms. Furthermore, it is awkward to hide away the Prolog behavior from the user who should ignore the underlying language.

However, Prolog is a viable option to implement algorithms. In fact, a LWB prototype has been written in Prolog, and a prototypic interface of the current LWB to SICStus Prolog has been tested. If required, it can easily be integrated into future versions.

#### Advantages of a Lex–Yacc Solution

Besides being the most widespread efficient tools for compiler construction, lex and yacc offer good flexibility and support for specific requirements as they always occur in complex systems. As for efficiency, flex allows, for instance, to control the degree of table compression and scanner optimization, if this ever becomes an issue. Furthermore, since they generate C-code, they fit well in systems for which other tools such as a GUI-builder are used.

There is also a disadvantage. Once the parser is generated, the accepted language is fixed. For instance, it is not possible to add or remove operators dynamically. New operators, however, can be mimicked by a prefix notation using a functional expression. Assume we need an additional binary operator. Then we choose an unused symbol, say `newop`, and formulate expressions with `newop` as follows:

```
> protect(newop);  # (forbid assignments to newop)
> s:= newop(a,b) -> newop(a,c);
```

### 3.3.2 Format of the Input File

Like the lex input file the yacc specification file is divided into three sections. The double percent signs serve again as separators. The percent sign is generally used in yacc as escape character.

*definitions*
`%%`
*grammar rules*
`%%`
*subroutines*

#### The Definitions Section

In the definition section, the tokens, the non-terminals, the operators, the operator precedence etc. is declared. The token and operator definitions form the interface to lex. Code blocks enclosed by lines starting with `%{` and `%}`) are copied verbatim to the generated C source file before the beginning of `yyparse()`. In the LWB, such a block is used to include the header files needed, and most importantly, the file yacctions.C ('yacc actions') with many action routines for the rules section below.

For all tokens and non-terminals which have an associated value, the C-type of this value has to be declared. For instance, the token `tSYMBOL`, which reports that a sequence of character is a valid LWB symbol, has simply the scanned character string as value. All such types must be listed in the definitions section in a `%union` declaration.

```
%union {       /* yacc stack type */
  char *str;
  int i;
  operation* pp;
  exprrep* e;
};
```

To declare the non-terminals and tokens and their types, the yacc keyword `%type` for non-terminals and `%token` for tokens is used, followed by an identifier out of the above union in angle brackets and by the name:

```
%type  <i> arglist
%type  <pp> lhs_expr

...

%token <str> tSTRING     /* tokens returned from lex */
%token <str> tSYMBOL
```

Table 3.3 and Table 3.4 list all tokens and non-terminals of the LWB parser with an associated value along with an interpretation of these values and an example. Operators are a special kind of tokens and have normally no associated value. They are equally declared in the definitions section. See 'Operator precedence and associativity' (cf p. 68) for the syntax. A description of the value passing mechanism is given in Sect. 3.3.4.

### The Grammar Rules Section

The grammar syntax for yacc is a simplified version of BNF (Backus-Naur Form). Each rule starts with a non-terminal symbol and a colon and ends with a semicolon. Alternatives on the right-hand side of a rule are separated by vertical bars '|'. For each alternative a separate action can be specified.

| | | | |
|---|---|---|---|
| *non-terminal* | : | *rhs1* | *action1* |
| | \| | *rhs2* | *action2* |
| | \| | *rhs3* | *action3* |
| | ... | | |
| ; | | | |

In the following example, neither the non-terminals `expr`, `binop`, `unop` (expressions, binary operators, unary operators) nor the tokens `tEQ`, `tIMP`, `tOR` (equivalence, implication, disjunction), etc. have an associated value. The action parts of the operator rules, however, generate the corresponding meta-code in postorder.

```
expr :
  ...
  |  constexpr
  |  binop
  |  unop
  ...
;

binop :
  expr tEQ expr          { code_generator::add(EQ_OP); }
  | expr tIMP expr       { code_generator::add(IMP_OP); }
  | expr tOR expr        { code_generator::add(OR_OP); }
  | expr tAND   expr     { code_generator::add(AND_OP); }
  ...
;

unop :
  tNOT expr          { code_generator::add(NOT_OP);}
  | tBOX expr        { code_generator::add(BOX_OP);}
  ...
;
```

### 3.3.3 Operator Precedence and Associativity

The operator precedence and associativity of the operators have been chosen according to the most common usage (if such exists). The only operators associated to the right are the classical and the linear implication. All other operators are associated to the left. The operator precedence (restricted to logical operators) is shown in Table 3.2.

| | | |
|---|---|---|
| 1. unary operators | classical<br>modal<br>multi-modal<br>temporal<br>linear | `~` (`not`)<br>`box`, `dia`<br>`box7`, `dia4`<br>`H`, `P`, `G`, `F`, `X`, `A`, `E`<br>`!`, `?` |
| 2. conjunctions | classical<br>linear | `&` (`and`)<br>`&` (`with`), `X` (`times`) |
| 3. disjunctions | classical<br>linear | `v` (`or`)<br>`\|` (`par`), `+` (`llplus`) |
| 4. | temporal | `U` (until), `B` (before) |
| 5. implications | classical<br>linear | `->` (`imp`)<br>`-o` (`llimp`) |
| 6. equivalences | classical<br>linear | `<->` (`eq`)<br>`o-o` (`lleq`) |

Table 3.2: Operator precedence.

For instance, the formulae

```
> ~a v b & c <-> ~a & b;
> a imp b imp c;
> a llimp b llimp c; # (linear logic implication)
```

are parsed as:

$$((\neg a) \vee (b \wedge c)) \leftrightarrow ((\neg a) \wedge b) \;,$$
$$a \rightarrow (b \rightarrow c),$$
$$a \multimap (b \multimap c).$$

The non-logical comparison operators `=`, `<>` etc. are non-associative (i.e. `a = b = c;` is not a legal input) and have the lowest priority. Integer multiplication (`mult`) and division (`div`) are treated as conjunctions, and integer `+` and `-` like disjunctions.

The binary and unary operators are declared in the definitions section by `%left`, `%right` and `%nonassoc`, which define at the same time their associativity. Operators are declared in increasing order of precedence (i.e. Table 3.2 is reversed). Operators on the same line are at the same precedence level. For instance,

```
%nonassoc tEQUAL tNONEQUAL ...
%left  tOR  tPAR ...
%left tAND  tWITH ...
%left tNOT tBOX  ...
```

declares the equality and inequality operator to be non-associative, all others to be left-associative, the negation to precede conjunctive operators, which in turn precede the disjunctive operators etc.

### 3.3.4 Value Passing

In the bottom-up parsing process, more and more complex structures are recognized up to full LWB statements. However, the parser is not only to answer the question whether or not a sequence of tokens is a valid statement but it has also to store and return a transformation of its input. In the case of the LWB, the result of the parser is usually a block of meta-code handed down to the interpreter. We refer to these results and intermediate results simply as 'values' which are associated to tokens and non-terminals. Note that the association can also be implicit; the generated meta-code, for instance, is stored in a global object.

#### Token Values

In many cases, we need more information than that a token has occurred. For instance, it is not sufficient *that* a string was scanned but we also want to know *what* string. The lexical analyzer `yylex()` returns either just the next token or pairs of the form $(token, value)$. The value of a token can be seen as an atomic intermediate result. The token itself is a constant, e.g. `tSTRING` is the integer `258`. If there is an associated value, it is passed through the yacc-variable `yylval` to the parser. Since there are different types of token values, `yylval` is declared to be of the union type described in Sect. 3.3.2. The token definitions and `yylval` make up the interface between `yylex()` and `yyparse()`.[2] Table 3.3 shows the LWB tokens which have an associated value.

#### Referring to Values

In bottom-up parsing, the action part of a rule usually synthesizes the results of the 'lower' levels to the result of the current non-terminal. Assume that `NT` is a non-terminal and $\mathtt{T}_1$, $\mathtt{T}_2$, $\mathtt{T}_3$ and $\mathtt{T}_4$ are other non-terminals or tokens through which `NT` is defined. That is, we have a rule

[2] Physically, the token definitions and the declaration of `yylval` are in the yacc generated file y.tab.h (renamed to yacc.C.h in the LWB ).

| token | type | meaning | example |
|---|---|---|---|
| tSTRING | <str> | the string | > "a string!"; |
| tSYMBOL | <str> | the symbol | > my_symbol; |
| dSYMBOL | <str> | the debug symbol | > p my_symbol; |
| tINTEGER | <i> | the integer | > 435; |
| tBOXNR | <i> | the index | > box5 a; |
| tDIANR | <i> | the index | > dia17 a; |

Table 3.3: Tokens of the LWB with an associated value.

```
NT  :
    ...
    | T1 T2 T3 T4   {action}
    ...
```

Then action can refer to the values associated with $T_1 \ldots T_4$ with $\$n$, where $n$ indicates the position of the token in the rule (i.e. $n = 1 \ldots 4$). Consider the concrete rule below: the value associated with the non-terminal `arglist` is referenced by `$3` (note that the literal tokens `'('` and `')'` are counted as well).

```
symbseq :
  symbol '(' arglist ')' { code_generator::add(INVOKE_OP, $3);
                           code_generator::add(IS_SYMBOL_OP);
                           $$=1; }
  ...
;
```

`$$` denotes the value of the non-terminal on the left-hand-side of the rule. `$$` is the explicit result and must therefore be of the type declared in the `%type` statement for this non-terminal. For rules without action parts the action { `$$ = $1;` } is implicitly assumed. In the example, the value of `symbseq` is set to `1`, saying that `symbseq` contains just 1 symbol if the displayed rule is triggered.

Table 3.4 lists all non-terminals of the LWB parser with an associated value ('#' means 'number of' or 'length of'). The rightmost column shows an example in the LWB input language.

### Embedded Actions

The value passing is somewhat special in the case of embedded actions. An action is said to be embedded if it is placed before the end of the right-hand-side of a rule:

*non-terminal* : (*rhs part1*) { *embedded action* } (*rhs part2*) {*action*};

Yacc replaces this rule by introducing a new non-terminal and a rule for it with an empty *rhs*:

| non-terminal | type | meaning (range) | example |
|---|---|---|---|
| `arglist`<br>`arglist_nonempty` | <i><br><i> | # arguments (0...)<br># arguments (1...) | `> f(a, b & c, d);` |
| `arglist_cons`<br>`arglist_nonempty_cons` | <i><br><i> | # arguments (0...)<br># arguments (1...) | `> array[2,3];` |
| `arrayexprlist` | <i> | # expr list (0...) | `> array(a,b,,,);` |
| `symbseq` | <i> | # sequence (1...) | `> protect(a,b);` |
| `substlist`<br>`nonemptysubstlist` | <i><br><i> | # subst list (0...)<br># subst list (1...) | `> s { a/p1, b/p2};` |
| `issilent` | <i> | 1 if silent proc def (boolean) | `proc : f() begin end;` |
| `arg_decl_list`<br><br>`opt_arg_decl_list` | <i><br><br><i> | # declared arguments (1...)<br># declared optional arguments (1...) | `> proc f(a,var b) begin ... end;`<br>`> proc f([a,b]) begin ... end;` |
| `lhs_expr` | <pp> | start operation of lhs | `> s[1] := a;` |

Table 3.4: Non-terminals of the LWB with an associated value.

*non-terminal* : *rhs part1* (*new non-terminal*) (*rhs part2*) { *action* };
*new non-terminal* : (no rhs) { *embedded action* };

The new non-terminal and the embedded action, respectively, can also have a result. The type of the result must be supplied explicitly since *embedded action* is not associated with a declared non-terminal. The embedded action can assign its result by using `$<type>$` instead of `$$`, and *action* can refer to this result by `$<type>n` instead of only `$n`. Embedded actions are often used to keep hooks to previously generated data when parsing flow-of-control statements, for instance, the *if–then–else* construct:

```
statement :
  ...
  | tIF expr           { $<pp>$=code_generator::reserve(); }
    tTHEN statement    { $<pp>$=code_generator::reserve(); }
    elsestat           { back_patch_if($<pp>3, $<pp>6); }
  ...
;
```

After the *if* condition (`expr`) the first embedded action at position 3 reserves room for a conditional jump to the beginning of the *else* block. The jump target, as well as the target of the unconditional jump at the end of the *then* block that skips the *else* block, can only be filled in at the very end of the construct. This technique is known as *backpatching*. It saves one compiler pass.

### 3.3.5 LALR(1) Parsing

LALR parsers, where LALR stands for 'Look Ahead Left Recursive', are special LR parsers. The (1) denotes that the lookahead is limited to a single token. LR parsing is an efficient, bottom-up syntax analysis technique covering a large class of context-free grammars. In particular, LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars exist. Yet, there are a few pecularities to be observed.

#### Left and right recursion

When a grammar is required to describe infinite structures, or finite structures of which the size is not known beforehand (e.g. lists), it usually contains recursive rules, in which the lhs non-terminal occurs also in the rhs.

LR parsers prefer — as suggested by the name — left-recursive formulations of rules, which they can handle more efficiently.

```
list:  list ',' item ;   /* left recursion */
list:  item ',' list ;   /* right recursion */
```

In the left recursive case the list is reduced after each item, whereas in the right recursive version, all preceding items and commas are pushed on the stack.

#### Shift/Reduce Conflicts

A shift/reduce conflict occurs when the parser can use a token either to complete the right-hand-side of a rule (reduce) or to shift it on the stack for the right-hand-side of another rule. Reduce corresponds to a left-associative behavior and shift, which is the default, to a right-associative behavior. E.g.,

```
a and f(a,b)[1];
```

is parsed as

```
a and (f(a,b)[1]);  # f(a,b) is shifted,
                    # and later used for reduction
```

and not as

```
(a and f(a,b))[1];  # f(a,b) is immediately used for reduction
```

#### Limited Lookahead

There are cases in which LALR parsers cannot distinguish clear situations due to the limited lookahead. The prominent case in the LWB language is the distinction of plain expressions and expressions on the left-hand-side of assignments:

```
a[2][1] ;          # display a[2][1]
a[2][1] := ...;    # assignment to a[2][1]
```

When parsing from left to right, it is not possible to decide which case we are in until the end-of-statement token ';' or the assignment operator `:=` is reached. The decisive token is far beyond the lookahead horizon. There are two solutions:

1. Reformulation of the language. In this case, a prefix version of the assignment (e.g. `assign(a[2][1], ...);`), shifting the decisive token to the left, would do.

2. Treating the cases in parallel and finalizing the generated data structures when the decisive token is read.

To preserve the common infix notation, the second option is applied for the LWB assignment. Expressions which can occur on the left side — basically symbols and indexed symbols (represented by the non-terminal `lhs_expr`) — are treated as right-hand-side expressions but the necessary information to transform them to left-hand-side expressions is continuously built up. See also p. 83, where the code generation for indexed assignments is described.

### 3.3.6 Parse-Time Errors

In this section the handling of errors occurring at *parse-time* is discussed. These are:

1. Plain syntax errors.

2. Context-sensitive errors.

'Ordinary' syntax errors are automatically recognized and handled as described below. Action routines, however, which check for context-sensitive inconsistencies, must raise a parse error explicitly.

For instance, writing two binary operators in a row is a plain syntax error. The ASCII LWB reports it as follows:

```
> a & & a;
parse error at '&':
a & & a;
    ^
```

A context-sensitive error, on the other hand, is for instance the double declaration of a local variable within a procedure definition:

```
> proc f() local a, b, a; begin end;
error: Symbol declared twice at 'a':
proc f() local a, b, a; begin end;
                     ^
```

The context here is the previously parsed list of local identifiers. The syntax itself, however, is fine. In general, the detection of context-sensitive errors at parse-time is tricky and therefore often deferred to runtime. The most important case is the argument-checking of built-in functions: it is the responsibility of the function to reject invalid arguments. There are also cases in which the checking could be done at parse-time, but in which it is deferred anyway, for the sake of a uniform treatment of 'interactive' parsing and procedure parsing. Consider, for example, the following statement:

```
> i + 3;
```

If `i` does not denote an integer, an error could be raised by the corresponding action routine. However, the same statement in a procedure definition is perfectly okay and the error detection *must* be postponed to runtime.

```
> proc f() begin i + 3; end;
```

Depending on the state of the session when `f()` is called, an error may or may not be raised. While `i` may have been unassigned at parse-time, it may denote an integer when `f()` is called.

### Syntax Errors

Whenever yacc detects a syntax error, the parser calls `yyerror()`[3] and enters the error recovery mode, looking for a state in which it can shift an `error` token. Any partially parsed rules are ignored. Essentially, the `error` token is used as synchronization point in the grammar at which it is likely that processing can continue. In the LWB, the parser picks up after the end-of-statement marker '`;`', i.e. in case of a parse error, anything up to the next '`;`' is discarded. The only `error` token in the LWB grammar is in the last alternative of the top-level rule of which the non-terminal is `lwbcommand`:

```
lwbcommand : /* nothing */
  |  lwbcommand statement  { ... }
  |  lwbcommand variacmd   { dev.flush(); }
  |  lwbcommand error tEOS { dev.flush(); yyerrok; }
```

In the error case, all input tokens are read and discarded until the first token that can follow the `error` token is encountered, here the end-of-statement token `EOS`. Normal parsing resumes only after the next `EOS`. With `yyerrork` the parser is told that error recovery is complete and that it should report subsequent errors.[4]

[3] `yyerror()` is defined in lex.l.

[4] Normally, it refrains from reporting another error until it has shifted three consecutive tokens without another error.

### Context-Sensitive Errors

When context-sensitive errors occur, which can only be detected by the action code, the macro `YYERROR` must be called to set the parser in the same error recovering mode as after an incorrect token. The above double declaration of local variables is caught as follows:

```
localvars :
   /* nothing */
  | localvars tLOCAL identlist tEOS
;

identlist :
  tSYMBOL                  { if (declare_local($1)) YYERROR;
                             delete [] $1; }
  | identlist ',' tSYMBOL  { if (declare_local($3)) YYERROR;
                             delete [] $3; }
;
```

The helper function `declare_local()` (defined in yacctions.C) does the actual check and the call of `yyerror()`.

### The Yacc Error Routine `yyerror()`

The error routine `yyerror()` has two functions:

1. Resetting the system and clearing the data built up so far.
2. Error string handling.

The data to be cleared consists only of meta-code of the erroneous statement. Since only the meta-code of correct statements is executed, erroneous statements will not produce side-effects; for instance, if a function call is part of the statement, it is not executed yet, even if the syntax of the function call itself is correct. `yyerror()` is either called automatically (syntax errors) or explicitly before the macro `YYERROR` (context-sensitive errors). Its task is also to display the erroneous token. See also A.3.2.

## 3.4 The Interpreter

The interpreter is the 'heart' of the programming language and of the treatment of user input. The LWB interpreter is called 'the LWB Abstract Machine'. It is abstract or virtual in the sense that the operations do not fit a particular computer architecture but are at an intermediate level. This is important when portability is a consideration. From the intermediate code, in the following called *meta-code* for short, the back end(s) of a compiler would generate the native code for possibly different target machines. The LWB Abstract Machine, however, interprets this meta-code, basically by executing corresponding C functions.

Beside a general overview, we discuss in this section a selection of tasks and problems related to code generation and interpretation of the LWB input language:

**Code Generation**

- Switching between 'interactive' parsing (immediate execution) and parsing of procedures (cf p. 86).
- Distinction between left-hand-sides (references) and right-hand-sides (values) of expressions (cf p. 83).
- Distinction between global (user) variables and local variables and parameters (within procedures) (cf p. 82). Scope resolution (cf p. 80).

**Interpreter**

- Evaluation of function names to internal functions or other LWB functions, call of these functions, argument passing (values *and* references) (cf p. 80).
- Maintenance of the runtime stack on which the meta-code operates. It stores the intermediate results and the activation record(s) of the currently active function(s) (call stack) (cf p. 79).
- Runtime error treatment (cf p. 86).

The technical aspects of the LWB Abstract Machine are described in the diploma thesis of Franz Achermann [1]. The explanations given here are complementary. A few parts which have changed or have been left out in [1] are discussed in detail, for example, the section on partial evaluation. Hints which should ease the orientation within the implementation can be found in A.3.3.

**Terminology**

- The term '*meta-code*' is used for the (sequence of) operations, also called primitives which are interpreted by the LWB Abstract Machine.

- The term '*code generation*' refers to the meta-code which is generated out of LWB language source code.
- The term '*global*' refers to symbols and function names at the user level, which are usually stored in the user module.
- The term '*local*' refers to symbols and function names which are declared inside a LWB procedure.

### 3.4.1 The Basic Design

**Interpreter Overview**

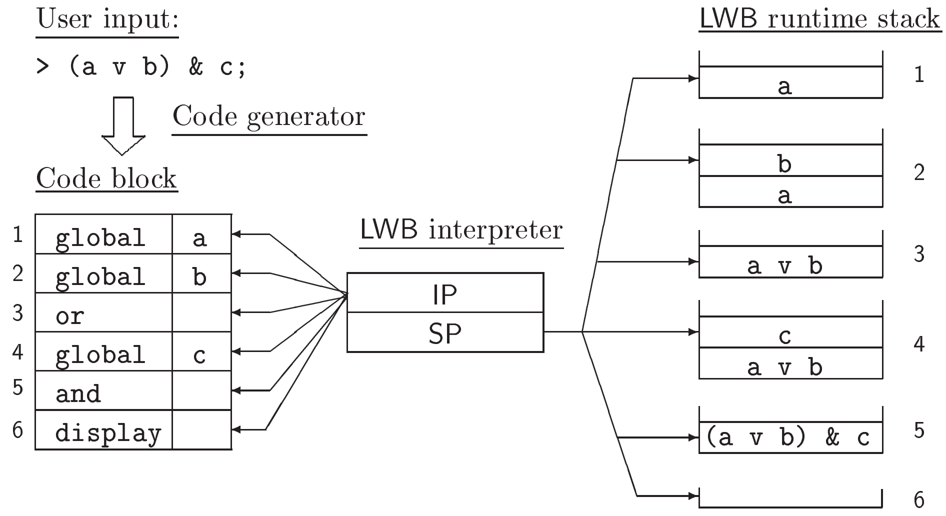



Figure 3.9: The LWB Abstract Machine: survey.

The LWB Abstract Machine is a stack-based machine, i.e. all instructions are performed on values on the LWB runtime stack. There are three classes of instructions:

1. Operations on LWB expression (logical operations, integer operations, etc.).
2. Operations on the runtime stack.
3. Control flow operations.

A (logical) expression is represented in meta-code in *postfix order* which reflects the way a stack-machine works: first, all arguments are pushed on the stack and then the operator instruction takes them down to build the corresponding data structure which in turn is pushed on the stack. To store the operands before the operator is also known as 'Reverse Polish Notation' (RPN).

When the LWB Abstract Machine is started, the control is transferred to a section of meta-code to perform the corresponding operations. In Fig. 3.9 the 6 stages of a simple interactive statement are depicted. Note that at the end of an interactive statement the stack must be empty again. The `IP` ('Instruction Pointer') indicates the instruction which is about to be executed. `SP` ('Stack Pointer') indicates the current stack position from where the instruction will fetch its arguments.

Needless to say, the pointer and runtime stack manipulations must be coordinated precisely — otherwise the LWB Abstract Machine is likely to crash very badly; filling up the memory because the stack is not properly cleaned is then a rather harmless problem.

### Instruction Execution Overview

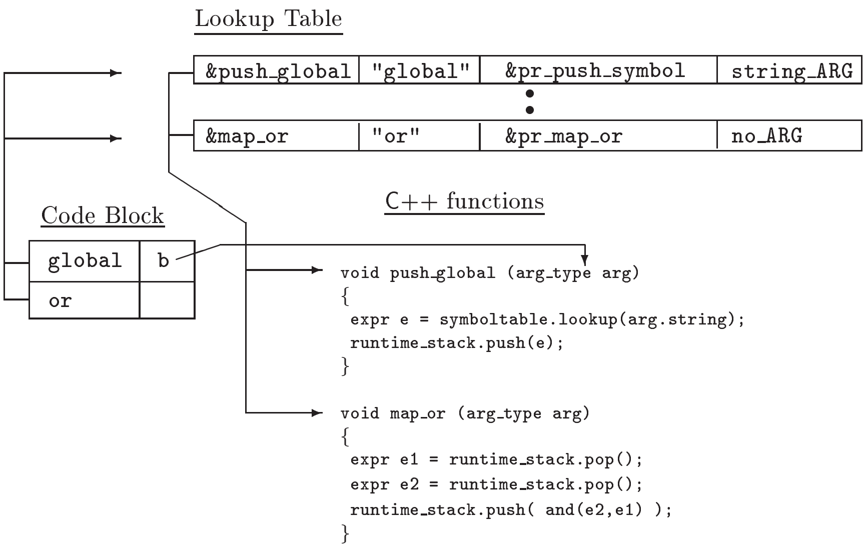



Figure 3.10: Instruction execution overview.

A graphical overview of an instruction execution is given in Fig. 3.10. The operation code (op-code) of a meta-code line is actually an index to a lookup table which basically contains function pointers. When the meta-code line is executed, the C++ function pointer corresponding to the op-code is called with the op-code argument as argument (if there is no op-code argument a default argument will be passed). For instance, when `global b` is executed, the C++ function `push_global` with string argument `"b"` is called, which simply pushes the corresponding symbol (expression) on the stack.

Note that the meta-code can also be interpreted in *print-mode* in which not the runtime function pointers of the table are used, but the 'print-time' function pointers (prefix `pr` in the figure). Each operator knows how to print itself, and so the LWB Abstract Machine can *disassemble* the meta-code (cf. [1], Sect. 7). Some operations have the same runtime behavior (i.e. function pointer) but a different print-time behavior, e.g., the jump operations of the *if* and the *while* constructs. Others print the same way and behave differently at runtime, e.g., the different operations which push symbols on the stack.

**The Runtime Stack**

The runtime stack is basically a stack of expressions to store the operands (intermediate results) for the operations to be executed.

The runtime stack also serves as call stack on which the activation records of the called functions are pushed (see Fig. 3.11). The parameter section contains the functor (`arg0`), the arguments passed to the function and the local variables, which can be treated like actual parameters. These are all expressions. The saved-machine-state section, however, consists of three pointers, of which the `ip` points to next instruction of the calling function, `argp` to the parameter section of the calling function and `fp` (the frame pointer) to the saved-machine-state section of the calling function. The temporaries section, which again contains expressions, reserves enough space for all intermediate results calculated during the function execution. The size of this section can be determined beforehand. For more details see A.3.3 and [1], Sect. 5.

$$
\begin{array}{l}
\left.\begin{array}{|r|l}\hline \text{arg0} & \leftarrow argp\\ \hline \text{arg1} & \\ \hline \vdots & \\ \text{arg}n & \end{array}\right\}\text{parameters}\\
\left.\begin{array}{|r|l}\hline \text{ip} & \leftarrow fp\\ \hline \text{argp} & \\ \hline \text{fp} & \\ \hline\end{array}\right\}\text{saved machine state}\\
\left.\begin{array}{|r|l} \vdots & \\ & \leftarrow sp\\ \hline & \end{array}\right\}\text{temporaries}
\end{array}
$$

Figure 3.11: Layout of an activation record.

## 3.4.2 Language Specialties

In this section, a few LWB specific features are mentioned which differ from other programming languages. References to the sections in which they are discussed in detail are given.

### Parameter Checking

Since the LWB language is untyped, the parameter checking is basically reduced to a checking of the correct number of arguments in a function call. However, if some parameter position is required to be referential (`var` keyword), we must ensure that the supplied (unevaluated) argument is a symbol and not a composed expression.

The parameter checking of global functions is carried out at runtime. In general it cannot be done at parse-time. Suppose the functions `a()` and `b()` are declared as follows:

```
> proc a( x )      begin return x; end;
> proc b( var x )  begin return x; end;
```

If `a` is assigned to the global symbol `f`, the function call `f(p v q)` is correct, while if `b` is assigned to `f`, it is not:

```
> f:== a;
  f( p v q );     # ok
        p v q
> f:== b;
  f( p v q );
ERROR in b: var parameter expects non const expression
```

Since the value of `f` at runtime can differ from the one at parse-time, the check must be deferred. Note that `f` needs not to be assigned at all, in which case a function expression is returned:

```
> unassign( f );
> f( p v q );
        f(p v q)
```

The deferring of the distinction between reference and value arguments implies that all arguments which can qualify as reference argument are pushed as such. The function has to convert them to values if only the latter are needed (cf. [1], Sect. 5.3 and 6.1.3).

The call of local functions is simpler: the function headers are known at the time when the call is parsed (at least a forward declaration must have been passed), and the parameter checking can be done at parse-time.

### Access to Non-Local Variables

Since procedures in the LWB programming language can be nested, a variable can be located in an outer scope (enclosing procedure). At runtime, the variable will be found in another activation record on the runtime stack. Whenever a procedure is entered (primitive `invoke`) the enclosed local procedures, which are part of the parameter section of the activation record as any other local variable, are equipped with an *access link*, a pointer to the current activation record. For instance, if two scopes in the source code have to be left to find the declaration of a variable, two access links have to be followed at runtime to find the corresponding activation record. This 'scope distance' and the position of an identifier within the scope (which becomes the offset within the activation record at runtime) are determined at parse-time and stored along with each command which accesses the identifier. See also Sect. 5.6 and 8.4 in [1].

### Optional Arguments

Value parameters at the end of the parameter list may be declared optional by embracing them with square brackets. If they are not supplied in the function call, they are initialized with the `NULLexpr` which can be tested inside the procedure with the `isnull()` user function.

```
> proc : f(a, [b,c])
  begin
    if isnull(b) then return a;
    if isnull(c) then return  a -> b;
    return a -> b -> c;
  end;
> f(p); f(p,q); f(p,q,r);
p
p -> q
p -> q -> r
```

### Range Extraction

Particularly useful beside the usual list operations is the possibility to extract whole (sub)ranges of lists:

```
> A:= [ true, false, p, box p, dia p, box dia p ];
        [true, false, p, box p, dia p, box dia p]
> A[3..5];
        [p, box p, dia p]
```

One margin may be omitted in which case the respective list boundary is taken.

```
> A[3..];
        [p, box p, dia p, box dia p]
```

The `extract` primitive is explained in Sect. 6.6.1 in [1]. Of course, `extract` takes a copy of the range and leaves the original list intact.

### 3.4.3 Code Generation

The code generation for the LWB programming language is largely straightforward (e.g. code for logical expressions). However, there are a few distinctions, often related to efficiency considerations, which complicate the code generation considerably. The most important distinctions are

| | | |
|---|---|---|
| global | $\Longleftrightarrow$ | local |
| rhs | $\Longleftrightarrow$ | lhs |

They are discussed in the following subsections. Since no case excludes any other, we often have four different meta-code sequences (global-rhs, global-lhs, local-rhs, local-lhs) for basically the same piece of source code, depending on where it occurs. So for some operations we need four op-codes. The actual complication, however, occurs at parse-time when the distinction and the proper code generation for these cases is carried out.

A simple way to inspect the meta-code to which a statement is translated is to put the statement inside a procedure and to display this procedure in debug mode. For instance the code for `a v b -> c` is:

```
>  debug( proc() begin a v b -> c; end );
FUNCTION ref(3)
L262992 begin   # proc()
L263004 global  a
L263016 global  b
L263028 or
L263040 global  c
L263052 imp
L263064 pop_expr
L263076 proc_end
```

The different classes of operations are discussed in [1], Sect. 6, in which also a complete list of the primitive operations is given in appendix B.

#### Global vs. Local

A major distinction is the one between global and local symbols and function names. A global symbol is more than a nonlocal symbol at the outermost scope. The main difference is that they are stored at a different place and that global symbols offer convenience features for interactive usage, such as evaluation and the fact that no declaration is needed.

| global | local |
|---|---|
| stored in symbol tables of modules | stored in activation records (runtime stack) |
| lookup needed when accessed | at a fixed offset |
| need no declaration | need a declaration (`local` keyword) |
| each time evaluated | never evaluated |
| return themselves if not initialized | return `NULLexpr` if not explicitly initialized |

A reason for the differences is that global variables may act either as object- or as metavariable while local variables are always metavariables which stand just for a value as all variables in a conventional programming language do. Since a particular global variable should not appear as both, object- and metavariable at a given state of a user session, evaluation ensures that once the global variable has been assigned, it will always evaluate to its assigned value even in expressions in which it has been used as object-variable:

```
> a:= c v d;
        c v d
> c:= m -> n;
        m -> n
> a;
        (m -> n) v d
```

The `c` within `a` evaluates to its new value when `a` is used again. Note that if `a` were a local variable it would still stand for the value assigned in the first line.

Although global variables behave exactly the same way inside a procedure, it is strongly recommended to use the more efficient local variables and value parameters for programming which need no evaluation.

### Rhs vs. Lhs

Another obvious distinction is needed between code for left-hand-side expression and code for right-hand-side expressions. It is mainly due to the fact that memory is shared which has to be split again when used on the left hand side.

In Fig. 3.12 it is the expression `a[1]` which must be treated differently: when used on the rhs, we keep on sharing memory and when used on the lhs, `a` must be copied before being accessed for assignment.

The difference between lhs and rhs usage of an expression is that in the lhs case the reference rather than the value of an object is required. This reference is then used to change the object for which purpose it may no longer be shared. However, if it has not been shared, the operation (e.g. assignment) can be done in-place and no copy is needed. Beside the assignment, the only LWB operations changing an operand are the list operations `push()`, `pop()` and `append()`, and the integer operations `inc()` and `dec()`. See also Sect. 6.4 'In-place Operations' in [1]. They copy precisely what is needed to avoid side-effects.

### Global Indexed Lhs

Particularly tricky is the case of a possibly multiply indexed global symbol on the left-hand-side of an assignment. The trouble starts with the inability of the parser to recognize whether or not the indexed expression is on the lhs:

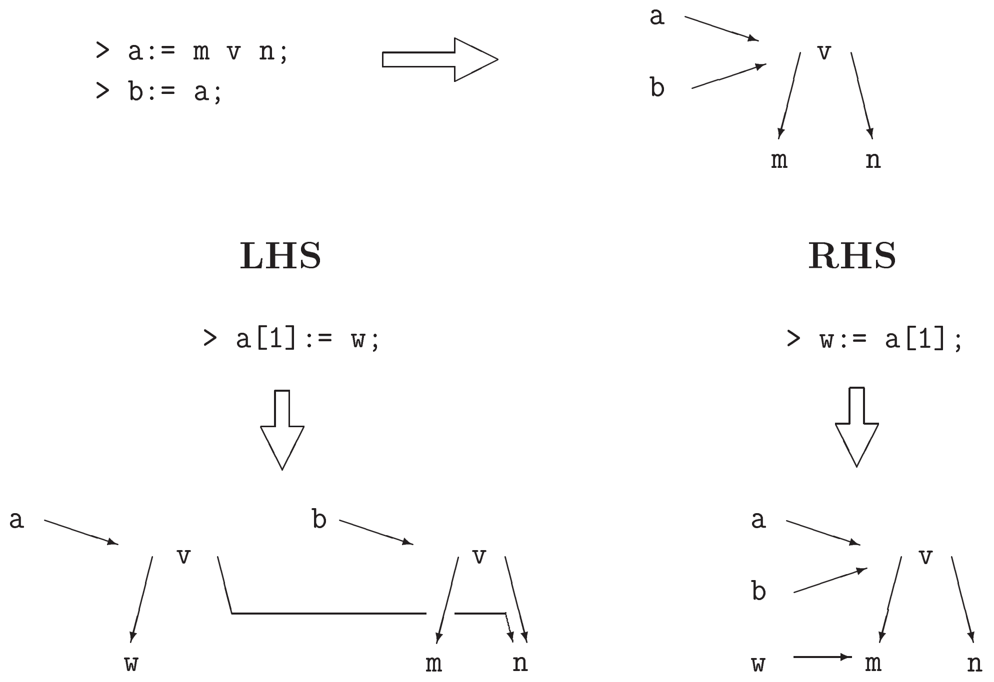



Figure 3.12: Rhs vs. lhs.

```
> a[1][2] := ...;   # lhs
> a[1][2] ;         # rhs
```

Due to the limited lookahead (cf. Sect. 3.3.5) the parser can only decide that it is a lhs when the assignment operator `:=` is encountered. Note that thanks to the functional notation of the other in-place operations in which the discerning token appears first, the distinction in the these cases is always clear. The assignment is even more unpleasant because the rhs of the assignment is expected to be executed first. The adopted solution is to parse the indexed expression always as rhs expression while keeping the necessary entry points to transform the rhs code to the corresponding lhs code if it turns out to be an assignment.

**Partial Evaluation:** It is obvious that an indexed global expression should never be fully evaluated neither in the rhs nor in the lhs case. Imagine

```
> A:= array[10000];
  ...
> s:= A[53] & A[49];
```

Then not the whole array should be evaluated (even twice in the rhs of the assignment) but only the 53$^{\text{th}}$ and the 49$^{\text{th}}$ element.

**Code Transformation:** In Fig. 3.13 the code for `a[i][j]` is shown and how it is transformed for lhs usage. The '`unique_..`' versions of the primitives ensure

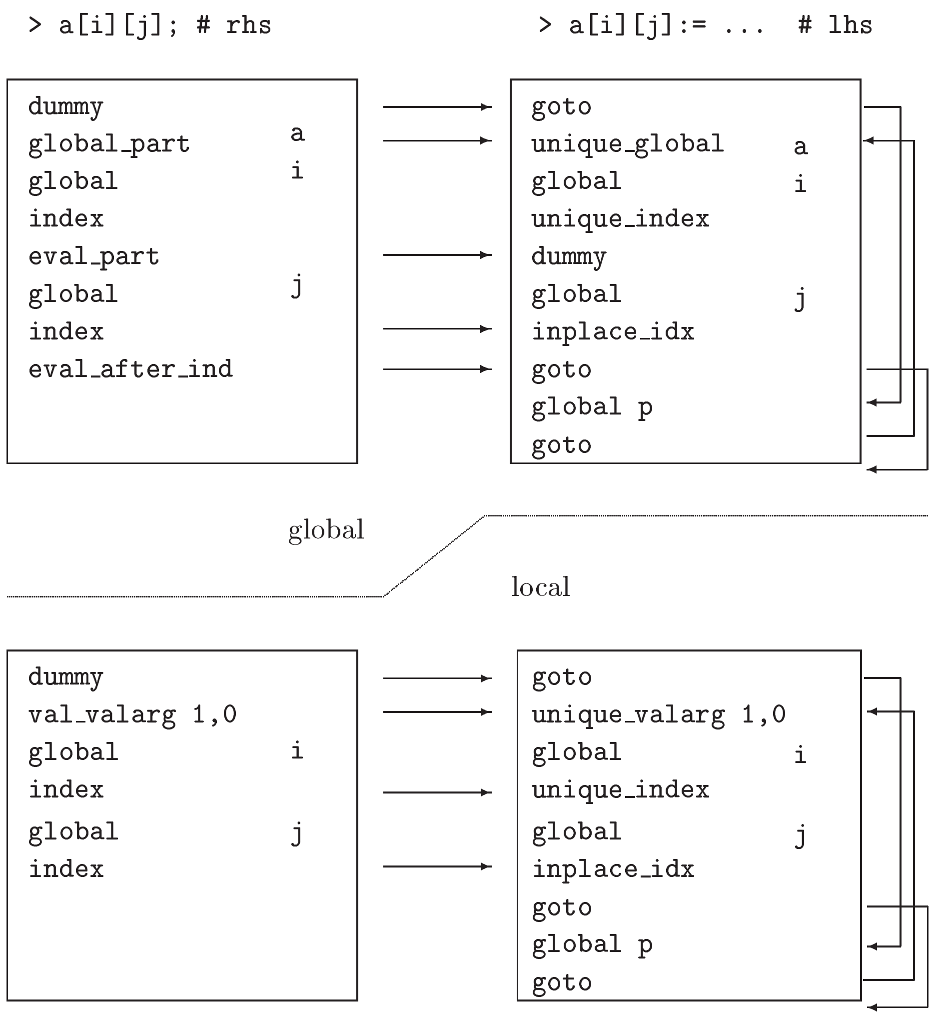



Figure 3.13: Code transformation (from rhs to lhs code).

that the respective node of the expression tree is 'uniquely' possessed, i.e. that it is copied if it is referenced by other expressions. The '`..._part`' primitives provide the partial evaluation and the `eval_after_ind` primitive (rhs only) does the full post-evaluation of the subexpression denoted by the indices. Note that if `a` is a local variable, no evaluation is needed at all. The `inplace_idx` primitive, finally, executes the assignment.

### 3.4.4 Interactive Mode

In interactive mode each statement is individually executed. This is the default mode when a user types in commands. However, execution starts only after a statement is correctly completed including the end-of-statement marker '`;`'. If a parse error is detected, only the meta-code generated so far is deleted and no side-effects occur.

Each statement is wrapped in an special procedure *input_proc* with no arguments and no local variables. Common procedures have always a return value: if there is no explicit `return`, the primitive `proc_end`, which terminates each procedure, pushes the `NULLexpr` as result. The *input_proc* on the other hand is not supposed to have a result and needs therefore a special terminator, the primitive `input_end`, which leaves the stack empty.

### 3.4.5 Runtime Errors

There are three kinds of runtime errors:

- Implicitly raised errors by the LWB Abstract Machine.
- Type errors raised by an argument check function of a built-in algorithm.
- Explicitly raised errors from within an LWB procedure by the `raiseerror()` statement.

An error is implicitly raised for instance when a `push` is tried on an expression other than a list or array:

```
> noList:== p & q;
> push( p, noList );
ERROR in push: list or array expected
```

Note that only at runtime it is possible to decide whether or not `noList` is an illegal argument to `push()`. : a list could have been assigned to `noList` by the time when the `push` is executed.

The second case occurs when for instance a linear logic formula is passed to the classical prover.

```
> load(cpc);
        cpc  user
cpc> provable( !p with q );
ERROR in provable: invalid argument 1
```

An example for an explicitly raised error is:

```
> proc: f(x)
  begin
    if cpc::length(x) < 4 then raiseerror("longer!");
  end;
> f(p & q);
ERROR in f: longer!
```

In each of the above cases, essentially the error routine of the LWB Abstract Machine is called, `lwbAM::error()`, which properly cleans the runtime stack; it deletes, for instance, all temporary data ensuring thereby that no garbage is left in memory. However, unlike parse-time errors which induce no side-effects at all, previously executed statements cannot be undone. Imagine for instance the statement

```
> s:= f1() & f2();
```

If `f2()` returns with a runtime error then the assignment is canceled and `s` keeps its previous while the side-effects of `f1()` will still be effective.

Within an LWB procedure the user can also intercept errors by the `catcherror()` statement. When an error is caught, the error message is stored in the global user symbol `error` and the execution can be resumed possibly depending on what error occurred. For example (the `f()` is taken from above):

```
> if  catcherror( f( p & q) ) = error then print( "oops!!");
oops!!
```

See A.3.3 for a discussion of the implementation.

### 3.4.6 Performance

The following tables gives a rough idea about the performance of the LWB Abstract Machine. The timings have been measured using the LWB 1.0 (compiled with Sun CC, C++4.0.1 option -O) on a 110 MHz microSPARC II CPU. The listed numbers are seconds; differences ± 0.03 (for tests during less than 2 seconds) are not significant. The source code for the tests is in the file efficiency.lwb.

#### List vs. Array, Global vs. Local

In Table 3.5 the performance of list and array operations is compared. The list (array) size is 9000. Since list elements cannot be accessed directly the list manipulations involving an index operation are much slower (growth: $O(n^2)$ vs. $O(n)$). However, the `foreach` loop is just as fast, and size affecting operations are much faster for lists (last line).

As long as no modules are loaded and the symbol table is small (lookup costs!), the difference between global and local lists (arrays) is negligible. But already when globals need a proper evaluation, the operations get slower (compare the $4^{\text{th}}$ and $6^{\text{th}}$ line).

| | | Array | List |
|---|---|---|---|
| init global by `for` loop | indexed assign | 0.23 | 3.32 |
| init local by `for` loop | indexed assign | 0.24 | 3.33 |
| `foreach` loop over global | reading | 0.05 | 0.04 |
| `for` loop over global | reading | 0.25 | 3.32 |
| `for` loop over local | reading | 0.18 | 3.24 |
| `for` loop over global, `A[i]` $\neq$ `eval(A[i])` | reading | 0.46 | 3.54 |
| creation | push and append | 31.09 | 0.21 |

Table 3.5: List vs. Arrays, size: 9000.

### While vs. Recursion

In Table 3.6 the performance of `while` loop solutions and recursive solutions of simple tasks is measured. As expected, recursive solutions are slower, due to the function call overhead. However, in cases in which the recursive solution is shorter as for instance a formula traversal (first line), the (more complicated) while solution is not much faster anymore.

There is one pitfall, however, for recursive programming. If large expressions are passed as values, any in-place operation (e.g. `pop()`) will cause a copy of the expression regardless of whether the original expression will still be used after returning from the recursion.

| | While | Recursion |
|---|---|---|
| flattening a formula (length $\approx$ 15000, depth 40) | 0.99 | 1.04 |
| reverting a list (length 9000) | 0.20 | 0.47 |
| sum(1..n) (n=20000) | 0.48 | 0.95 |

Table 3.6: While vs. Recursion.

### For vs. While

`For` loops are more efficient than `while` loops (Table 3.7).

| | For | While |
|---|---|---|
| empty loop bodies (20000 iterations) | 0.24 | 0.54 |

Table 3.7: For vs. While.

### Argument Passing

In Table 3.8 four different ways of changing a value by a (local) procedure are compared: 1. The value is stored in a global variable. 2. The value is passed as value argument the changed value is returned as the function result. 3. The value is passed as reference argument. 4. The value is stored in a nonlocal variable (i.e. an access link is needed). It turns out that the variants are equivalent.

| | global | by value | by reference | nonlocal |
|---|---|---|---|---|
| value change (20000 calls) | 2.97 | 2.33 | 2.34 | 2.45 |

Table 3.8: Argument Passing.

### Comparison with Maple V

The following tables present the timings for similar procedures in the Maple programming language. Maple 5.3 has been executed on the same machine using the programming constructs coded in efficiency.maple. Tests have been carried out only for the cases in which a similar construct is available.

When comparing the results, we have to consider the following points:

- Maple employs a garbage collector which is launched every 250,000 words of allocated memory. This may result in much bigger time differences (mostly when operating with large lists or arrays) from run to run. The time of the shortest of several runs has been taken.
- Maple supports integers of arbitrary length. Therefore, integer operations are probably a little bit slower.
- In Maple the memory sharing is almost strict, i.e. each expression is stored just once in memory.

| | | Array | List |
|---|---|---|---|
| init global by `for` loop | indexed assign | 0.75 | 230.8 [5] |
| init local by `for` loop | indexed assign | 0.85 | — |
| `foreach` loop over global | reading | — | 0.10 |
| `for` loop over global | reading | 0.78 | > 1000 |
| `for` loop over global, `A[i]` $\neq$ `eval(A[i])` | reading | 0.77 | — |

Table 3.9: Lists and Arrays (Maple), size: 9000, cf. Tab 3.5.

| | While | Recursion |
|---|---|---|
| sum(1..n) (n=20000) | 2.95 | 5.82 [6] |

Table 3.10: While vs. Recursion (Maple), cf. Tab 3.6.

[5] `For`-loop combined with `subsop()` command. Creation using the `seq()` command, however, is *very* fast.

[6] If the stack size (`limit stacksize`) is not high enough a segmentation fault occurs.

## 3.5 Symboltables and Modules

Large collections of algorithms are usually divided into modules or packages which can be loaded individually. In the Logics Workbench, a natural partition is given by the set of contained logics, and in fact, the LWB 1.0 provides one module per logic. It is conceivable, however, that special sets of logic functions form a separate module, especially if they belong to an external contribution. The module concept allows structuring the user name space and to have the logic functions available 'on demand'.

Of course, it is possible that several logic modules are active at the same time; this is a prerequisite for the LWB to be a platform for comparisons of different logics. To facilitate such comparisons, it is most natural that similar algorithms of different logics have the same name at the user level. For instance, throughout the LWB the decision procedures are called `provable()` (and the dual `satisfiable()`). However, this requires a name resolution mechanism when more than one logic is loaded. A default rule picks the 'obvious' module if the user enters only the function name (see 3.5.2). So when a new module is loaded, it is possible that some functions are hidden in the sense of the default rule. But the user can always address a function of a particular module using the module operator '::'; for instance, the consistency check for intuitionistic theories is called with `ipc::consistent(..)` (provided the module `ipc` is loaded).

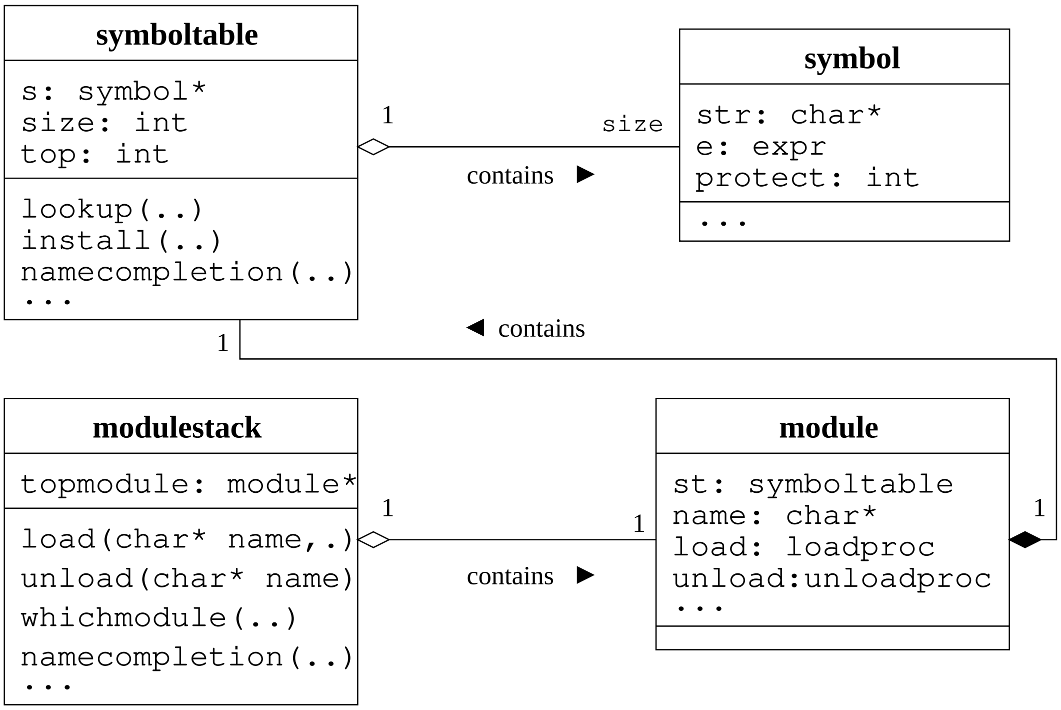



Figure 3.14: Symboltable and module classes.

### 3.5.1 The Symboltable

The symboltable can be viewed as the connection between the user symbols and the internal part of the system; it relates the symbols to something 'meaningful'. In this sense, it is a part of the interface to the user and belongs therefore to the LWB parser library. The same holds for the module classes which are built on top of it. A symbol has a 'meaning' if it is used as a metavariable, that is, if some expression has been assigned to it before. In the case of the user module (see 3.1.1), the symbols stand normally for user-defined formulae or user-defined procedures, while in a logic module, the symbols represent typically function descriptors which contain a function pointer to the corresponding logic library. The retrieval of a value of a symbol is referred to as 'evaluation' (see 2.2.1) where the access of a built-in function is just a special case (see 2.4.2).

The main method of the symboltable class `symboltable` is the lookup method (see Fig. 3.14). It checks to see whether the table contains a symbol with a given character string. If there is no such symbol, a new one is inserted with the install method. If the symboltable is full, it is automatically expanded. In any case the lookup method returns the corresponding symbol to the caller.

#### Name Completion

A very convenient feature of the symboltable is the name completion. The symboltable is searched for a symbol with the given string as initial part. If there is none, the empty `NULL` string is returned, and if there is at least one, the common initial part of all such symbols is returned. Imagine, for example, that the following two symbols are in the table:

```
provability
provable
```

Assume further that the table contains no other symbols starting with `pr`. Then the completion of `pr` is

```
provab
```

That is, the name is completed up to the character from which the completion candidates differ. If the name is preceded by a module identifier and the module selection operator, the completion is searched only in the symboltable of corresponding module. The name completion is available in the input regions of the XLWB (see Fig. 4.4), where the user can press the `Tab` key behind the word to be completed. If the word has a completion, it is automatically replaced.

#### The Problem of Unused Symbols

It may happen that the only reference to a symbol is the one from the symboltable. Such symbols might be considered as 'garbage' which should be removed. However, the situation is not as clear cut. Imagine that the a symbol has a value. If we remove the symbol (simply because it does not occur in another formula), this value is lost. A reasonable definition of an 'unused' symbol is the following:

1. The symbol's reference count[7] is 1: Then the table is the only object which references the symbol, and it is not contained in another expression.

2. The symbol is not assigned: It is not used as a metavariable and stands just for itself. In particular, it is not the symbol for a built-in function.

3. The symbol is not protected: It is not a constant, as for example `true`.

Since a symbol cannot know where the last reference comes from, it is the responsibility of the symboltable to carry out a garbage collection of unused symbols. Although the LWB 1.0 implements no such operation (so far, the number of unused symbols had no significant influence on the performance), the options are briefly discussed.

- When the table is full: The loop which copies the old array to a larger new array could ignore the unused symbols instead of deleting them individually.

- When a name is looked up: Any implementation of the lookup method visits a number of symbols, at least when the table becomes fuller (conflicts, if a hashing technique is applied). The visited symbols which are unused could be deleted.

- When the system is idle: When there is no input from the user, the garbage collection could be triggered periodically.

In the first two cases a garbage collection would be done 'on the fly'. This is potentially dangerous because of a technical detail: while the interpreter is running, not all expressions are referenced immediately after creation,[8] so symbols may seem unused although they are not.

Note that a frequent removal of unused symbols is not necessarily good, because the symbols may only temporarily be unused. The user can always clean the entire module stack using the `reset` command, possibly preceded by a `save("`*`filename`*`")` to save the assigned symbols to a file.

---

[7] See Sect. 2.1.3 for a description of the reference counting.

[8] For instance, if the code for `a & b` is executed, the `a` is pushed on the runtime stack as a temporary expression. In this case the runtime stack does not increment the `a`'s reference count since it (correctly) assumes that the conjunction will reference it.

**Implementation:** The present implementation of the symboltable class is very simple. The table consists of an array of `symbol` pointers which is scanned by `for`-loops in reversed order. Since symboltables are usually accessed during the interactive dialog, efficiency is not very important. Moreover, the symboltables of the logic modules are not large. As for the user symboltable, the reversed search ensures that the most recently entered symbols are quickly found. However, if it seems necessary, the implementation can be replaced, for instance by a hashing technique, although the display, the enlargement, and the name completion still require a linear traversal.

### 3.5.2 The Module Stack

Modules can be loaded and unloaded, where the set of loaded modules is organized as a stack. Since each module has a symboltable of its own, symbols, and especially function names, may occur several times in the stack of loaded modules. Beside the loading and unloading, the symbol lookup is the main task of the module stack. In particular, when no module is explicitly specified, it applies a default module selection rule. The module stack provides the following additional services:

- Bringing a lower module to the top: The top module is of course favored by the default selection rule; the user can omit the module prefix (module name plus '`::`') for functions of the top module.
- Name completion in a specific module or in the whole stack.
- Resetting the stack: The stack is completely emptied. It is the caller's responsibility to re-install at least the user module.

#### Default Rule for Module Selection

If a symbol exists in several loaded modules, the default rule (implemented in the method `whichmodule()` of the module stack) selects the topmost module in which the symbol occurs. This corresponds to a nested scope situation in which a symbol is taken from the innermost scope in which it is defined. You can also think of the module stack as a *PATH* environment variable of an operating system which is searched for executable functions.

If the symbol is not defined at all, the bottom module, which is always the user module, is returned by default. All new symbols are therefore entered in the user module[9] which cannot be unloaded or moved to the top. See also Sect. 3.1.1.

#### Loading a Module

A module can be viewed as an object with optional 'constructor' and 'destructor'. More specifically, the module can provide an optional load procedure and an unload

[9]See for instance the operation `push_val_of_global` in runtime.C.

procedure (function pointers `load` and `unload`). The load procedure may create any auxiliary, module-specific data structures or may even start up a helper process, for instance a Prolog. Conversely, the unload procedure should undo such preparations.

When a module is loaded, the module stack queries the module data in a global module description list[10] and pushes a new instance of `module`. Normally, a function pointer for a load procedure (`loadproc`) is given, and the module constructor calls it with the module's symboltable as argument. The general task of the load procedure is to install the module specific functions in the module symboltable. In fact, the LWB logic modules do not require any other preparations, and therefore they provide no unload procedure (`unloadproc`). See A.3.4 for a typical load procedure of a logic module. The return value of the load procedure indicates whether the initialization was successful and the module is ready for use, or whether the module installation failed, e.g. because a helper process did not come up.

### Module Extension by the User

When a user works often with a specific module, he may wish to extend it with functions of his own written in the LWB input language, or simply with some frequently used formulae. The module stack provides a hook so that such extensions are loaded along with the module. After the successful initialization by the load procedure, the new module is installed on top of the current module stack. Then a second initialization is done: Assume that *mod* is the name of the new module. If the file

`$LWBHOME/modules/`*mod*`/`*mod*`_init.lwb`

exists,[11] it is automatically parsed, and the contained LWB statements are executed. All user-defined extensions can go to this file. If the function names and metavariables in this file are prefixed with the module selection operator, they are installed in the symboltable of the new module rather than the user module. In this way, the definitions of the user extension are removed when the module is unloaded again.

[10] See A.3.4 for a short discussion of the module description list.

[11] `$LWBHOME` is the environment variable that indicates the home directory of the LWB installation.

# Chapter 4

# A User Interface for Propositional Logics

The graphical user interface plays a crucial role considering the aim of the LWB to be used in education. Having a powerful language and a flexible parser the LWB offers already a great deal of user-friendliness: especially the possibility to assign long formulae or lengthy function names to a metavariable or the possibility to write little helper functions alleviate the set-up of a customized environment. The XLWB enhances this user-friendliness by introducing sessions, which allow for instance to suspend the current work and to continue at a later time. The user can freely navigate within a session, can copy & paste, and can save a lot of typing by modifying existing input regions.

The concept described below is not restricted to a logic system but can be viewed as a general front-end to command driven mathematical software. In fact, it has been inspired by commercial dialogue-oriented math packages such as Maple V [13] or Mathematica (version 2.0 or later) [65].

### General Remarks

- In this chapter we will focus on the concept of the user interface, rather than the technical aspects of the implementation. However, it is worthwhile mentioning that one spends often a lot of time on little problems which attract only attention if they are *not* solved.

- A large part of each user interface does never change once it has been created, for instance, the menus or the arrangement of widgets within others. This part, in the following called *static*, will not be discussed in detail; it is straightforward and can easily be constructed using a GUI builder.[1]

- More interesting is the concept of the *dynamic* part of the XLWB, namely

[1] In fact, the next version of the user interface will partly be built using Motifation, a GUI tool developed at the University of Paderborn.

the session window containing the input and output regions. The number of regions can increase and decrease and the regions themselves can grow and shrink, output regions can be hidden and so on. Some special features offered by the regions will also be mentioned.

## 4.1 Motif

As a proposed standard for graphical user interfaces, Motif has been implemented on a wide range of computer platforms from large IBM mainframes right down to the PC. Most important for the LWB is, that it is the designated standard for UNIX platforms.

Motif is a set of guidelines which specify how a graphical user interface should *look and feel*. So in the first place, Motif is a user interface *specification* rather than an implementation (see the Motif Style Guide). But Motif is also a toolkit from OSF (Open Software Foundation) which is based on the X Toolkit Intrinsics (Xt). Xt provides an object-oriented framework for creating reusable, configurable user interface components called widgets. It includes functions for X Windows manipulation as for instance the setting of resources on widgets or the dealing with colormaps, events etc.

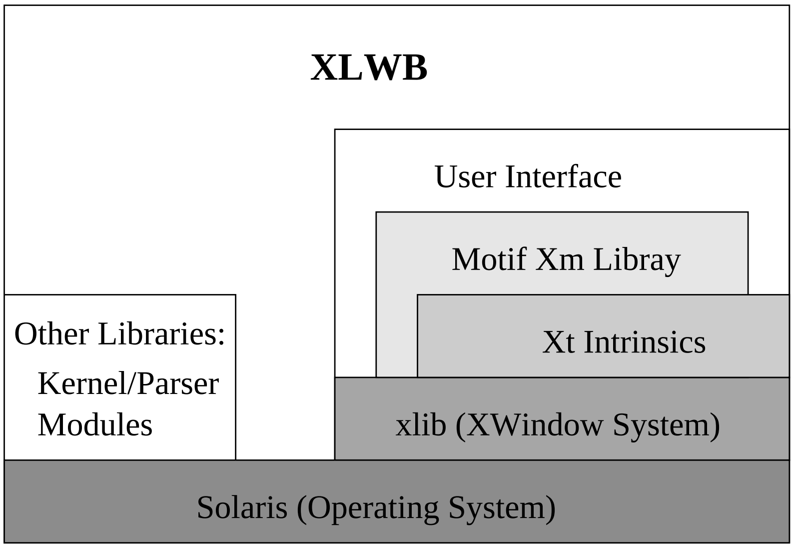



Figure 4.1: XLWB library model.

The Motif Xm library defines a complete set of widget classes, which is sufficient for most user interface needs. While the usage of Xt functions is inevitable, direct calls to Xlib functions should be reduced to a minimum. See Fig. 4.1.

### 4.1.1 Events and Callbacks

X user interfaces are *event driven*. Once the windows have been created, the control is transferred to a central event loop which analyzes the stream of events created by the keyboard, by mouse movements and mouse clicks. The application has the possibility to react to certain events within certain widgets by registering *callback* functions which are 'called back' by the event handler when the specified events are observed.

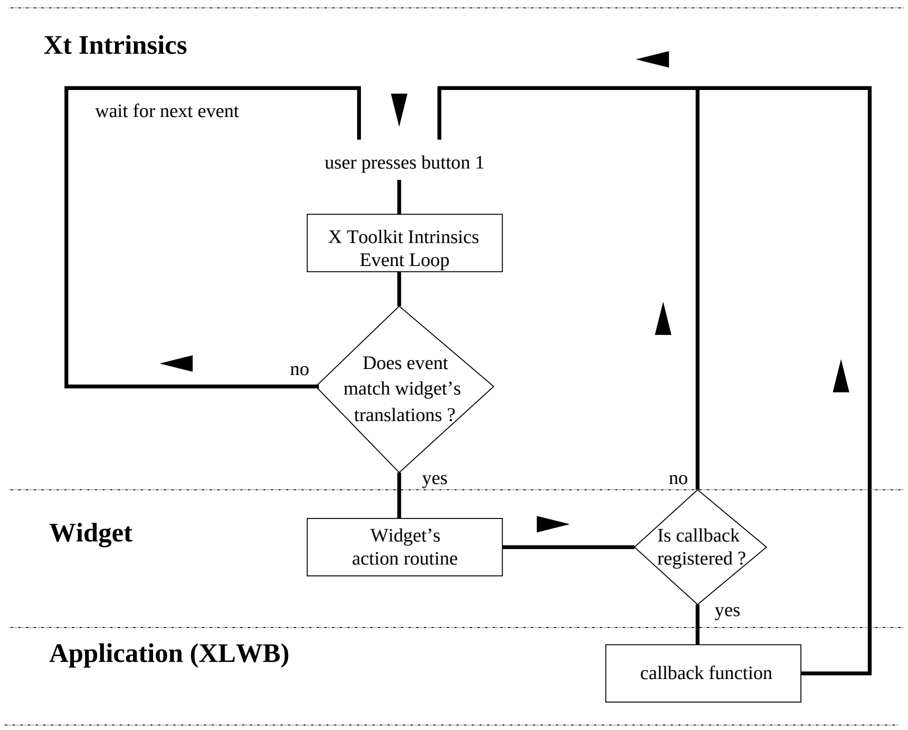



Figure 4.2: Event loop and callbacks.

In Fig. 4.2 the event handling is illustrated. The widget's translation table registers the widget's interest in a particular type of event. When Xt receives an event that happened in the widget's window, it tests the event against the translation table. If there is no match, the event is thrown away. If there is a match, the event is passed to the widget and an action routine is invoked. The action routine may perform a function internal to the widget (such as changing the widget's appearance by highlighting it). Then, if the action is associated with a callback resource, it checks to see if a callback function has been registered for that resource, and if so, it dispatches the callback.

A thorough introduction into Motif can be found in the *Motif Programming Manual* [40] and into Xt in the *X Toolkit Intrinsics Programming Manual* [55].

## 4.2 XLWB Sessions

An XLWB session consists of the input and output regions which form the dynamic part of the LWB. It is dynamic in the sense that neither the size nor the number of regions is fixed, that is, existing regions can grow or shrink or even be deleted and new regions can be appended or inserted at any position of the session. Also dynamic is the possibility to hide all output regions (for demonstrations) and to let them reappear one by one.

There are three main ideas behind the session oriented XLWB:

1. Convenience for the user: Minimizing the typing needed to accomplish a task.

2. Possibility to suspend the current work and to resume it at a later time.

3. Creation of sample sessions or assignment sessions for students.

An exemplary sample session can be found in Sect. 3.1.3.

### 4.2.1 Input and Output Regions

Both, input and output regions are multi-line textwidgets, where the output regions are not editable. Each input region, however, is a little text editor. The following features are supported:

- Multi-line editing: It is particularly useful when LWB procedures are to be defined, if desired along with comments and examples. See e.g. Fig. 4.3.

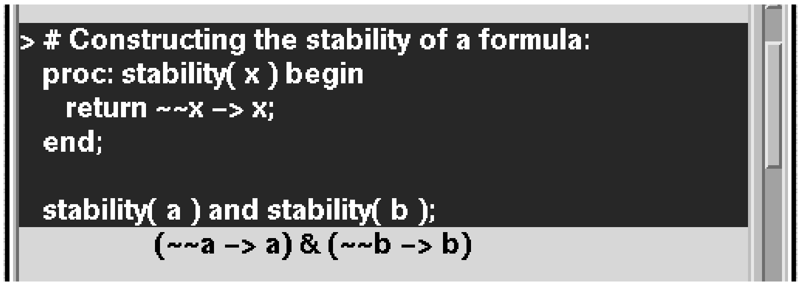



Figure 4.3: Multi-line editing.

- Automatic word wrapping: A new word wrapping takes places when the main window changes its width.

- Automatic scrolling if the cursor moves out of the clipboard.

- Support of Emacs control sequences:

| | |
|---|---|
| `Ctrl-a` | beginning of line |
| `Ctrl-e` | end of line |
| `Ctrl-d` | delete next character |
| `Ctrl-k` | kill to end of line |
| `Ctrl-w` | cut clipboard |
| `Ctrl-y` | paste clipboard |
| `Ctrl-j` | newline and indent |

- Name completion: Beside the function names of the loaded modules any user variables or function names are immediately available for name completion after their first usage. See also Sect. 3.5.1.

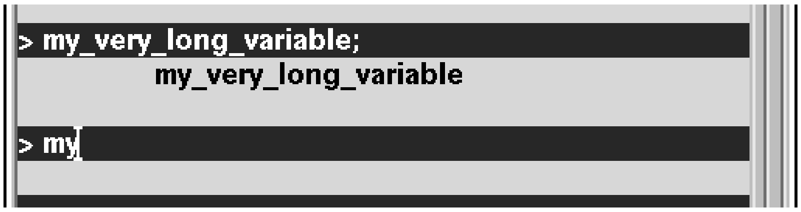



Figure 4.4: Name completion in the XLWB.

If in Fig. 4.4 the user presses `Tab` at the cursor position, the name `my` will be completed to `my_very_long_variable` (provided that no other names start with `my`).

- Graphical error indication: In case of a parse error, the incorrect token is highlighted and can be typed over without repositioning of the cursor.


Figure 4.5: Graphical error indication.

See also 'Parse Error Localization' on page 63.

- Electric parentheses: Whenever the user types a `)`, } or `]`, the corresponding opening parenthesis `(`, { or `[` is highlighted for a short moment in the preceding text of the current input region. This is most useful if complicated formulae are entered.

### 4.2.2 Navigation

The navigation between the regions is straightforward and is mostly done by mouse movements, scrollbar adjustments, and mouse positioning. The current or active input region has a blinking cursor which can be moved for instance by clicking on any position in any input region.

The '`Up`' and '`Down`' cursor keys move the cursor from one input region to the one above and below respectively, while '`Shift-Up`' and '`Shift-Down`' move the cursor within an input region up and down.

| Key | Between Regions |
|---|---|
| `Up` | Move to the previous input region |
| `Down` | Move to the next input region |
| `Enter` | Evaluate region & move to the next |

| Key | Within a Region |
|---|---|
| `Shift-Up` | Move to the previous line |
| `Shift-Down` | Move to the next line |
| `Shift-Enter` | Insert a new line |

The usage without `Shift` has been chosen for inter-region navigation because very often the input regions consist of a single line. If a key is 'overloaded' then the `Shift` version refers to the expected action within a region.

### 4.2.3 The Widget Tree

Each widget is of a specific *widget class* (type). A concrete widget is an *instance* of its class. All widgets have a name (not necessarily unique) and have one direct parent. So an X user interface is hierarchical and has a tree shape. A particular widget can be identified by giving the path from the root. Often the parents take responsibility for general tasks of their children. There are also widgets which do only provide services and have no window associated, e.g. the `ApplicationShell` widget which serves as the top-level widget of the application and which is invisible since it is overlaid by the main widget of the application.

Figure 4.6 shows the widget subtree for the XLWB main window. The full widget tree can also be obtained by the tool editres, which is supported by the XLWB.

### 4.2.4 Customization

#### XLWB Resources

Widgets are designed so that many of their properties can be changed by the programmer but also by the user (even at run-time). The public widget variables are commonly referred to as *resources*. The most important resources of the XLWB can be changed interactively in the 'LWB resources' window, see Fig. 4.7.

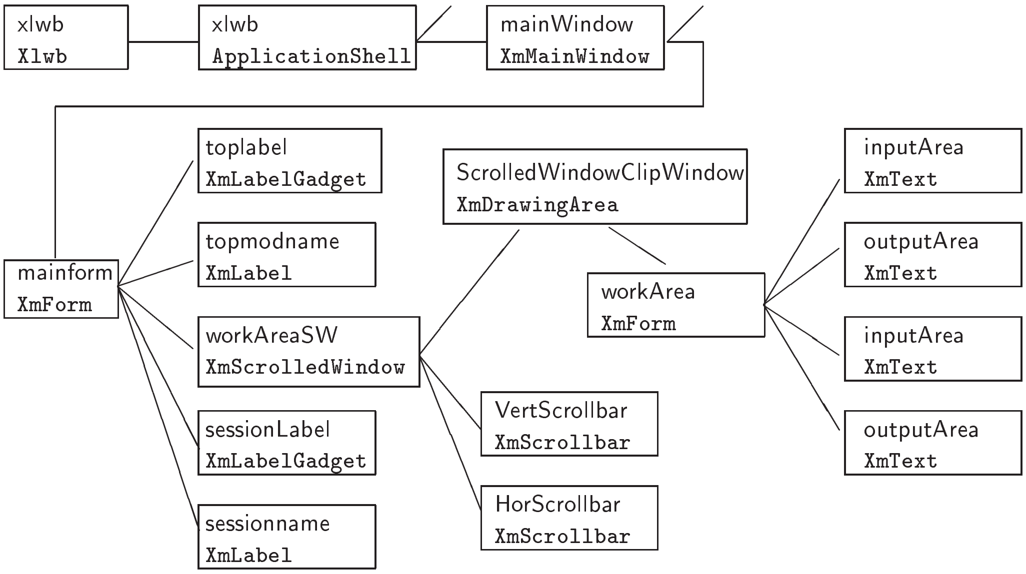



Figure 4.6: Main window widget names and Motif class names.

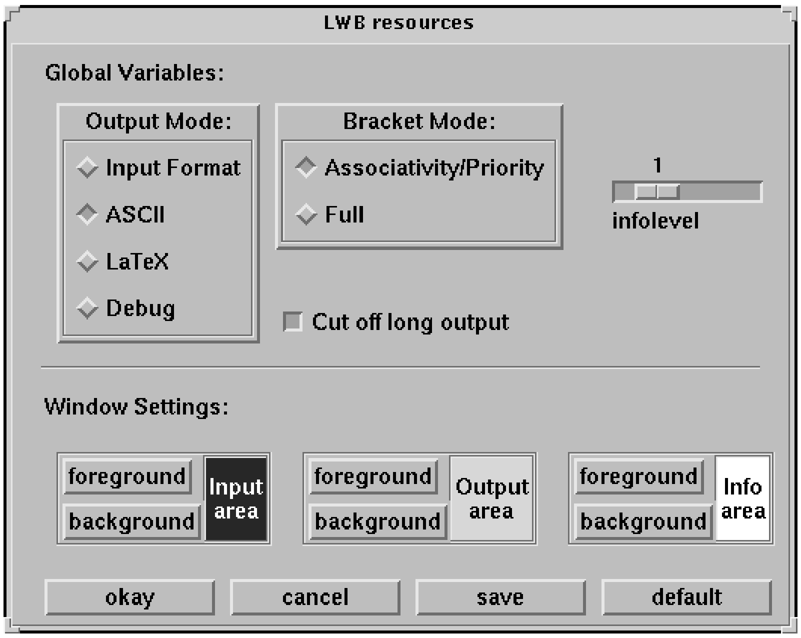



Figure 4.7: XLWB resources window.

When the user presses the `save` button, the settings of the resources window will be stored to the file .xlwb in the current working directory. The next time the XLWB is started, .xlwb is automatically read and the resources are set accordingly. If no file .xlwb is found (the user may also have a 'central' .xlwb in his home directory rather than in the working directory), the so-called *fallback* resources apply, which ensure a reasonable program operation.

### General Xt Resources

If the user knows the name or the class of a widget, he can change basically all its resources. To have them active the next time the XLWB is started, they are best stored in the file .xlwb as well (above the resources stored by a `save` button activation). The widget names and the names of the supported resources can be determined using the tool editres (see the section above).

Example: If a user wants to change the '`file`' menu entry to '`Datei`', the following line can be prepended to .xlwb:

```
Xlwb*file.labelString: Datei
```

Note that the asterix '`*`' matches any widget names in between. To be explicit, we could also write

```
Xlwb.xlwb.mainWindow.menubar.file.labelString: Datei
```

By using the class name instead of the widget name, it is also possible to set a resource for all widgets of a particular class. For instance,

```
Xlwb*XmPushButton.background: red
```

would set the background color of all pushbuttons to red (including the above `file` menu entry). By convention, names with the first character capitalized denote class names rather than instances. Accordingly, `Xlwb` denotes the class of the application while `xlwb` is usually the instance name (but it could for instance also be `testxlwb`).

### Attachments

In the XLWB only the resources essential to the graphical layout are hard-coded, mostly so-called attachments, which define the position of widgets with respect to others [40, 8.4]. The four sides of a widget can be attached to other widgets, e.g. we can specify in terms of attachment resources that a widget $C$ is below a widget $A$ and to the right of a widget $B$. Another resource is the distance (offset) from the neighboring widgets. It is important to have the children positioned relative to one another and relative to specific locations in the parent, in order to preserve a similar layout in case the user chooses a larger base font for the user interface.

### 4.2.5 Implementation

The XLWB 1.0 is entirely written 'by hand'. Most used widgets are encapsulated in C++ classes; the key classes and the widgets they handle is shown in Fig. 4.8. While a line on the class side means class derivation, a line on the widget side means parent–child relation.

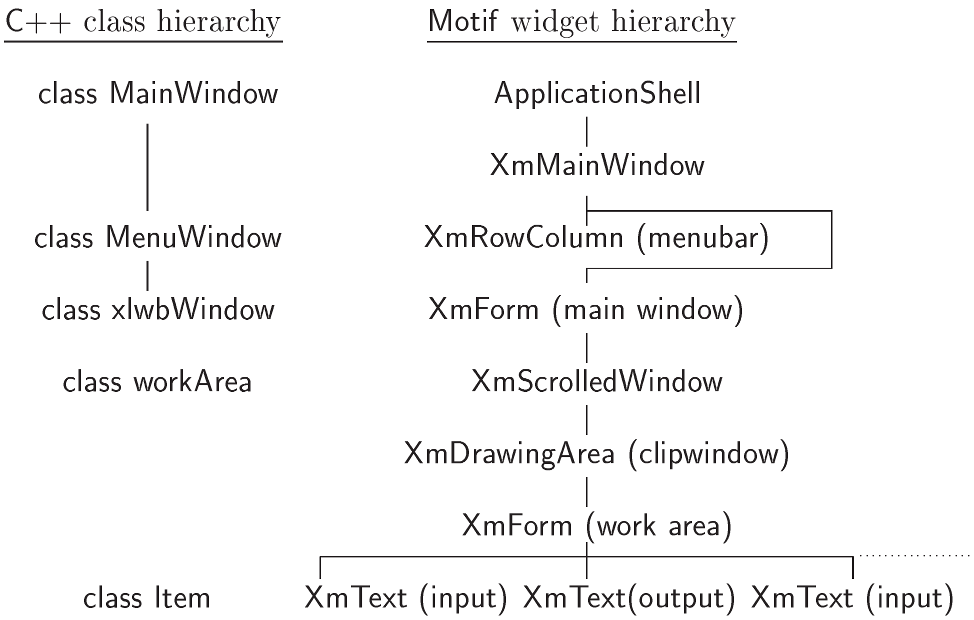



Figure 4.8: Class and widget correspondence.

The class `workArea` is a (private) data member of `xlwbWindow`. The class `Item` represents the list of pairs of input and output regions. `workArea` keeps a handle to this list.

## 4.3 The Interface to the Kernel

The interface to the kernel basically consists of:

1. Sending the content of the current input region to the parser.
2. Displaying the result in the current output region.

There are a few minor tasks discussed in the following sections.

### 4.3.1 LWB kernel $\Longrightarrow$ XLWB

The direction from the kernel to the XLWB deals with the piece of information that the kernel is to display at the user interface. If the XLWB has a special representation for some information, the following technique has been applied:

1. Introduce a new function header taking the information as argument.
2. Implement the function twice, once for the ASCII LWB and once for the XLWB.
3. Link the ASCII LWB version of the function to the ASCII LWB and the XLWB version to the XLWB.

So each new user interface has to implement this (small) set of special information functions (beside the standard output devices). The decision which implementation to take is deferred to link-time. Basically, the same technique is applied for the global meta-devices (see Sect. 2.3.1).

The ASCII LWB versions of the functions including the declaration of the global meta-devices are in plainversion.C, and the XLWB versions in parserconnection.C.

The following information may be differently displayed from one user interface to the other:

1. The result of user input: Results, warnings and algorithmic information are written to global meta-devices, see the section on the LWB output 2.3. While the ASCII LWB has only one real device to which all meta-devices are mapped, namely `stdout`, the XLWB implements two real devices:
   (a) `workwindevice` for the results (and errors/warnings). This device writes to the current output region.
   (b) `statuswindevice` for status information and algorithmic information. This device writes to the 'LWB infos' window.

   Both devices are buffered; they display the content of the buffer only if it is full or if an explicit 'flush' occurred. This avoids flickering of the screen if the output comes in small pieces.

   The meta-devices are bound to the XLWB devices at link-time. See Fig. 2.12
2. Parse error display: The XLWB version of `showparseerror()` highlights the erroneous token (see Fig. 4.5).
3. The user changes a global variable using the `set()` command instead of changing it in the 'LWB resources' window. The XLWB version of `globalvarupdatehook()` updates the resources window with the new values.
4. The user changes the stack of loaded modules by using the `load()`, `unload()` or `topmod()` commands instead of using the corresponding `module` menu entries. The XLWB version of `moduleupdatehook()` displays the new top module (a label on the main window) and updates the pull-right menus for the `module-unload` and `module-topmod` menu entries.
5. Some algorithms estimate their progress in percent. The XLWB version of `showpercentage()` displays the percentage in a slider on the 'LWB infos' window.

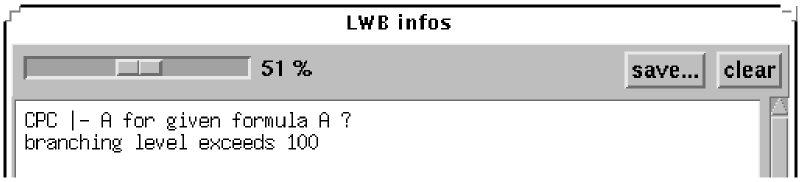



Figure 4.9: The percentage slider.

### 4.3.2 XLWB $\Longrightarrow$ LWB kernel

This direction of the interface is much simpler since the XLWB is on top of the rest of the application (see the library model Fig. 4.1) and can therefore use all the global kernel and parser variables and functions (for instance the global module stack (`ms`)).

The main interface is the evaluation of an input region when the user presses '`Enter`'. The content of the input region, which is a string since an input region is a text widget, is then sent to the parser using the overloaded function

```
int yyparse(char *s);
```

See the section 'Input from Strings' 3.2.3.

If other actions affecting the kernel can be expressed by an LWB command, this version of `yyparse()` can be used by passing the corresponding LWB command as string argument. For instance the `file-read` and `file-save` menu entries are implemented in this way. On the one hand, this saves a re-implementation of the action and on the other hand, it ensures that really the same operation is executed.

#### Other Differences to the ASCII LWB

**The `main()` routine:** The `main()` routine of the XLWB (Main.C) is hidden in a library. It is generic for all user interfaces based on the set of C++ classes (library libMB.so) which wrap the most important widgets. It initializes the application and enters then into the event loop. So at some point of the application initialization, the XLWB still needs to initialize the kernel (setting up the LWB Abstract Machine, initializing the module description list). See the function `initkernel()` in parserconnection.C.

**Interrupt handling:** Another difference is the interrupt handling which is also done in the ASCII LWB from within `main()`. Instead of once per session, the interrupt handler is installed each time the above `yyparse()` is called. If an interrupt is caught while `yyparse()` is active, one has to take care that no X commands are interrupted and possibly mangled with new X activity (which happens, for instance, if the interrupt handler displays a little message). So the execution of the handler is deferred to a safe moment (after the flush of the devices). The interrupt handling for the XLWB is also implemented in parserconnection.C.

## 4.4 LWBproof_wish

A separate part of the LWB user interface is the proof display tool, LWBproof_wish, ('LWB proof window shell'). It is independent of the XLWB since it is a separate process which is started by the XLWB (or the LWB) when the user asks for a proof.[2] An (incomplete) list of offered features:

- Display of the proof in a tree style.
- Highlighting of the principal formulae (red) and, if desired, of the side formulae (blue).
- Possibility to click down subproofs in which we are not interested.
- Hidden subproofs can be reconstructed step by step.
- It can be clicked on a rule label and the corresponding rule scheme appears in a little window.

If the LWBproof_wish is not present, the user can still look at proofs in a ASCII representation with indentations. An example is given in Fig. 4.10 where statistics about the proof have been cut out. The same proof using LWBproof_wish is shown in Fig. 4.11.

```
cpc> proofoptions( "indent" );
        true
cpc> proof( (a & b v c) & (a v d) -> a v d );

r ->            ==> (a & b v c) & (a v d) -> a v d
l &             (a & b v c) & (a v d) ==> a v d
w               a & b v c, a v d ==> a v d
r v             a v d ==> a v d
l v             a v d ==> a, d  [1.1,1.2]
ax      1.1       a ==> a

ax      1.2       d ==> d
```

Figure 4.10: LWB proof with indentations.

LWBproof_wish is a part of the diploma thesis of Peppo Brambilla [11]. A comprehensive description can be found there.

[2] At present, only classical proofs can be stored and displayed.

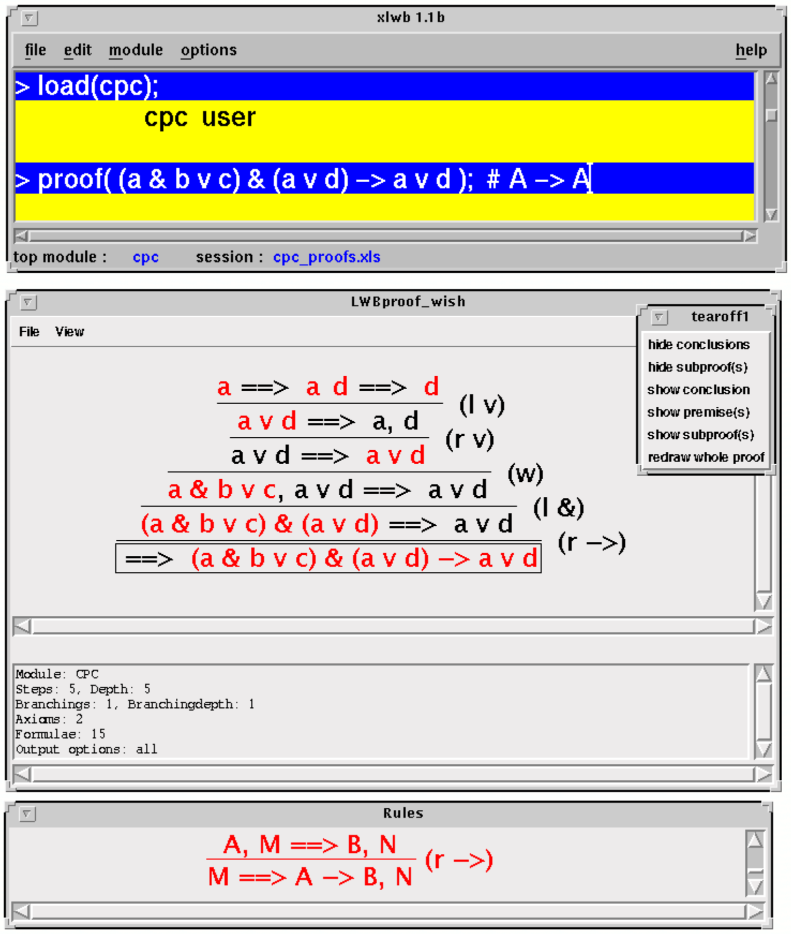



Figure 4.11: LWB proof in LWBproof_wish.

# Chapter 5

# Distributed Propositional Proof Systems

## 5.1 Introduction

The LWB presented so far is tailored for sequential computing and one-processor systems. In this chapter it is shown how the possibilities of a distributed environment can be exploited.

Special emphasis is put on non-classical deductive systems and proof search in these. The satisfiability problems of the considered modal logics are PSPACE-complete. These PSPACE problems seemed to be a promising field for distribution, and in fact, we will show that we can improve the performance of the algorithms significantly for certain formula classes. It was important to find a solution which keeps the costs of transforming a sequential algorithm into a distributed one low. The simplicity of the transformation allowed the consideration of several logics. It also underlines the practical relevance of the results; they do neither depend on a tuned hardware environment nor on a highly specialized software.

On the practical side, an extension dLWB (the 'd' stands for 'distribution') of the described Logics Workbench was produced. It takes advantage first of the modular structure of the LWB, which made the necessary adaption very easy, and second of the well-known and widespread PVM ('Parallel Virtual Machine'), which provides the basic features for a distribution of large tasks on several processes. Unlike the standard LWB, the dLWB is an experimental research prototype which is not intended for public use.

A fundamental preliminary work was the establishing of a set of reasonable benchmark formulae which have different characteristics and which are hard enough. The method described in the following does not rely on the form or any other feature of a certain formula class.

#### General Remarks

It is important to point out that there are many possible directions of investigation when we move to the field of distribution or parallelization. An interesting question is for instance, what the right granularity of parallelization is for a given problem. Maybe a coarse distribution to a set of multi-processor workstations, where a fine parallelization takes place, is ideal. Another interesting problem is load-balancing, that is, to ensure that the available resources are equally used and that the computation is distributed as long as possible.

There is an imminent danger that the focus of interest shifts more and more to performance issues, which play a much bigger role than in the sequential case. For instance, the intensity of communication is often a critical factor. It may also happen that new bottlenecks arise when the problem size is increased. Moreover, the success of the distribution technique depends heavily on the underlying hardware; for instance, the communication costs differ in an order of magnitude between a network linking workstations and an internal bus linking processors.

The aim of the following investigation is to show, whether and where there is a *general potential* for distribution in the context of non-classical proof search. It was not the intention to develop a specialized parallelization technique, tailored to the available hardware environment.

The solution is a competitive approach with a minimum of communication. In many cases it works so surprisingly well that it is hard to imagine how a technically more complicated cooperative solution might keep up with it.

### 5.1.1 PVM

The Parallel Virtual Machine PVM [24] is a software system that permits to link a heterogeneous collection of (originally) UNIX computers together via the local network, so that a user program can view it as a single large parallel computer. PVM has been developed within a collaborative research project of the Oak Ridge National Laboratory and three universities. It provides a functionally complete message-passing model. Briefly, the most important principles upon which PVM is based are:

- User-configured host pool: The user selects the set of machines for a given run of the program. The host pool may also be altered (adding/deleting of machines) during operation.

- If required, specific tasks can be positioned on specific machines.

- The unit of parallelism in PVM is a *task* (usually a UNIX process), an independent sequential thread of control alternating between communication and computation.

- Explicit message-passing model: The computational tasks cooperate by explicitly sending and receiving messages to and from one another.

One of the key advantages of PVM is that it keeps the cost of distributed computing low by making use of the existing hardware.

The PVM system is composed of two parts (see also figure 5.1).

1. A daemon, called pvmd, that resides on all computers of the virtual machine: It is a background process that serves as a message router and controller. It creates and binds TCP sockets to talk to tasks and other pvmd's and opens an error log file.

2. A library of PVM interface routines: It contains user-callable primitives for message passing, spawning processes, coordinating tasks and modifying the virtual machine. This library is linked to the (sequential) task programs which make up the distributed application.

### 5.1.2 Computing Models

Some applications are parallelized along their functions, which means that each task performs a different operation. This method is often called *functional parallelism*. The more common method is the *data parallelism*, in which the tasks are the same, but each one is working only on a part of the data. This is also referred to as the SPMD (single-program multiple-data) model of computing. Of course these methods can be combined by splitting the algorithm in functional chunks (components) which work internally according to SPMD.

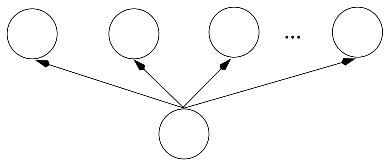
### 5.1.3 Parallelism and Classical Logic

Many classical parallel theorem provers have evolved from successful sequential ones, for instance, PARTHEO [9] (PARallel THEOrem prover) from SETHEO [48], ROO [21] (Radically Optimized OTTER) or the METEOR [4] family of provers. Other known parallel provers are PARTHENON [9] and MGTP [30] (Model Generation based Theorem Prover). All these systems treat classical first order logic, and most of them have in common that they apply only OR-parallelism and that they run on a parallel architecture. The published speed-ups are roughly linear. However, also in classical logic, they are sometimes far below linear and sometimes startlingly superlinear. A survey of parallel inference systems (including parallel versions of Prolog) can be found in [60].

There are also statistical approaches which investigate the theoretical potential for speed-ups. W. Ertel [23] introduces the notion of 'Random Competition' and suggests a 'Competitive Parallel Execution Model' which is very close to the design of the dLWB presented in the following. Based on this model, Ertel can compute theoretical speed-up figures if the runtime of a random search is considered as a stochastic variable with a certain probability density.

In addition, there are models which try to explain the speed-up that can be observed with the classical propositional satisfiability problem. They are based on the assumption that some parts of the search tree contain a higher proportion of solution leaves than others. Such models are reviewed and criticized by Goerdt and Kamps in [34]. To check the models, parallelizations of simple versions of the Davis-Putnam procedure have been used on random formulae in clausal form. It is known that the probability to obtain hard formulae (i.e. formulae at the 'threshold' of being unsatisfiable) is related to the ratio of the number of clauses and the number of variables. In [34] experiments have been carried out with formulae having 600 to 800 clauses of length 3 in 200 variables. The best speed-ups seem to occur in a region where the formulae are 'quite satisfiable'.

Unfortunately, the situation in non-classical logics is more complicated. Giunchiglia and Sebastiani [32] tried to extend the classical fixed clause-length model to the modal logic K. Hustadt and Schmidt [44] criticized their approach because it creates too many trivially satisfiable or unsatisfiable formulae. They suggest an improved generation of random modal 3CNF formulae of modal depth 1 (!), and they have indeed obtained challenging problems by increasing the number of clauses.[1] However, the complexity of these formulae is essentially classical; satisfiable K formulae of modal depth 1 have simple and 'flat' Kripke structures of the form:




A restriction to such formulae is certainly not an appropriate test for a general modal prover.

## 5.2 The Design of the dLWB

The overall design of the dLWB follows the master-slave paradigm. The master process controls the input and output, starts the slave processes, equips them with the initial data, and collects the results. As for the actual work part, the dLWB is very much like a SPMD application; all machines run the same slave executable. There is one difference to most such applications: it is often unnecessary to wait for the results of all tasks. In such cases late tasks are killed by the master. Note that a master-slave application of this kind can easily be converted into a pure SPMD application by combining the master and the slave in a single executable (with an external flag indicating the role it plays).

The design of the dLWB was influenced by the fact that the environment of the application is based on a network (as opposed to a multiprocessor topology), where

[1] The number of variables is fixed; they suggest 4 and 6.

a large granularity leads to better performance. So only solutions minimizing the frequency and volume of communication were considered.

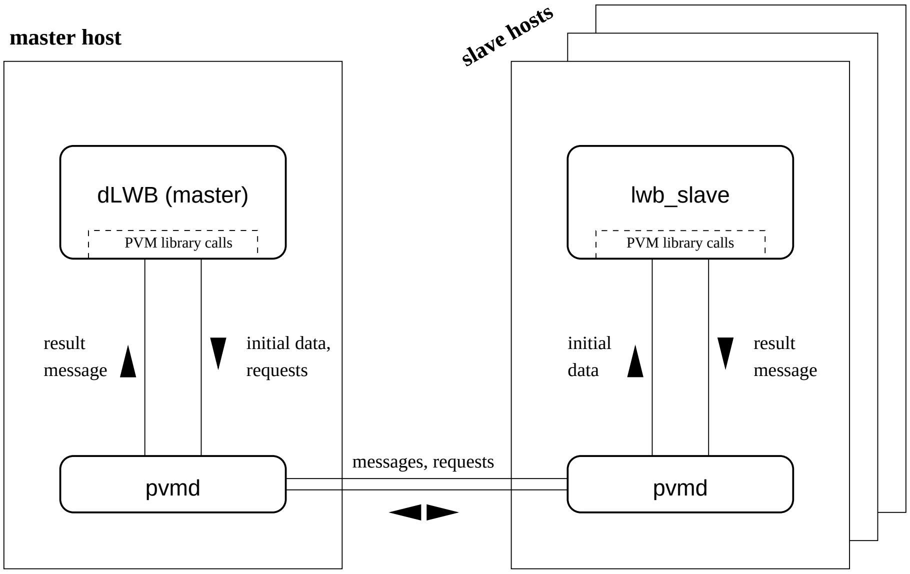



Figure 5.1: The dLWB layout.

The present dLWB has the following characteristics (see also figure 5.1):

- No fixed assignment of tasks to a specific machine (except the master which runs on the machine where the dLWB is started).[2]
- Only the master starts tasks, that is, the fork-and-join is restricted to the master process.
- No communication among the slaves.
- After having spawned the slaves, the master process waits in a blocking receive for the results of the slaves. This adds no load to the master machine, which can therefore host a slave as well.
- The virtual machine is not reconfigured dynamically. The dLWB determines the number of available hosts and places exactly one slave on each.

The slaves do not start subtasks in a divide-and-conquer manner because of the extreme ratio between the possibilities to split (up to several hundred thousand during a proof search) and the number of workstations (less than hundred); the workstations are immediately 'used up'. This is a fundamental difference to divide-and-conquer by recursive function calls of which the number is virtually unlimited.

[2]Currently, PVM does a round robin assignment, starting with the next host in the table (i.e. the relative performance or the machine load are ignored).

It may be successful to have several slaves on a single processor machine: we can mimick a kind of breadth-first search by using implicitly the job scheduling of the operating system. The task switches, however, lead to a significant overhead. To exploit this speed-up potential, the development of a (limitedly) multi-threaded application, in which only so-called light-weight processes are employed, would be appropriate.

## 5.3 The Modal Logics

We briefly introduce the syntax and the sequent calculi of the three well-known modal logics for which we will later present a distributed decision procedure. For a general introduction to these logics including the semantics we refer to [29, 35, 18].

### 5.3.1 The Language for $\mathsf{K}$, $\mathsf{KT}$, $\mathsf{S4}$

**3.1 Definition** The set of propositional variables Var is defined as:
$\mathrm{Var} := \{p_i \mid i \in \mathbb{N}\}$.

**3.2 Definition** The set of formulae Fml is inductively defined as follows:

1. The propositional constants $\mathsf{true}$ and $\mathsf{false}$ belong to Fml.
2. All propositional variables belong to Fml.
3. If $A$ and $B$ belong to Fml, then $\neg A$, $\Box A$, $\Diamond A$, $A \wedge B$, $A \vee B$, $A \rightarrow B$, $A \leftrightarrow B$ belong to Fml.

**3.3 Convention** In the following we use the capital letters $P, Q, \ldots$ as metavariables for elements of Var, and $A$, $B$, $C, \ldots$ as metavariables for elements of Fml.

### 5.3.2 Basic Definitions

**3.4 Definition** The length of a formula is inductively defined as follows:

1. If $A$ is $\mathsf{true}$ or $\mathsf{false}$, then $\mathrm{length}(A) := 1$.
2. If $A$ is a propositional variable, then $\mathrm{length}(P) := 1$.
3. If $A$ is a formula $\neg B$, $\Box B$, or $\Diamond B$, then $\mathrm{length}(A) := \mathrm{length}(B) + 1$. If $A$ is a formula $B \wedge C$, $B \vee C$, $B \rightarrow C$, or $B \leftrightarrow C$, then $\mathrm{length}(A) := \mathrm{length}(B) + \mathrm{length}(C) + 1$ .

**3.5 Definition** The depth of a formula is inductively defined as follows:

1. If $A$ is $\mathsf{true}$ or $\mathsf{false}$, then $\mathrm{depth}(A) := 0$.
2. If $A$ is a propositional variable, then $\mathrm{depth}(A) := 0$.
3. If $A$ is a formula $\neg B$, $\Box B$, or $\Diamond B$, then $\mathrm{depth}(A) := \mathrm{depth}(B) + 1$. If $A$ is a formula $B \wedge C$, $B \vee C$, $B \rightarrow C$, or $B \leftrightarrow C$, then $\mathrm{depth}(A) := \max(\mathrm{depth}(B), \mathrm{depth}(C)) + 1$.

**3.6 Definition** The modal depth moddepth of a formula is inductively defined as follows:

1. If $A$ is $\mathsf{true}$ or $\mathsf{false}$, then $\mathrm{moddepth}(A) := 0$.
2. If $A$ is a propositional variable, then $\mathrm{moddepth}(A) := 0$.
3. If $A$ is a formula $\neg B$, then $\mathrm{moddepth}(A) := \mathrm{moddepth}(B)$. If $A$ is a formula $\Box B$, or $\Diamond B$, then $\mathrm{moddepth}(A) := \mathrm{moddepth}(B) + 1$. If $A$ is a formula $B \wedge C$, $B \vee C$, $B \to C$, or $B \leftrightarrow C$, then $\mathrm{moddepth}(A) := \max(\mathrm{moddepth}(B), \mathrm{moddepth}(C))$.

### 5.3.3 Sequent Calculi

The following two-sided sequent calculi form the basis of the proof search of the LWB 1.0 in K, KT, and S4.

**3.7 Convention** In the following $\Gamma$, $\Delta$, $\Sigma$, $\Phi$, $\Psi$, and $\Pi$ are used for multisets of formulae. The set notation is used for multisets (e.g. $\in$ for the element relation, $\cup$ for union etc.) and is defined as usual. In the rules of a calculus we also write $A, \Gamma$ for $\{A\} \cup \Gamma$ and $\Gamma, \Delta$ for $\Gamma \cup \Delta$. $\Box\Gamma$ is used for the multiset $\{\Box A \mid A \in \Gamma\}$, and $\Diamond\Gamma$ for the multiset $\{\Diamond A \mid A \in \Gamma\}$. The cardinality of the multiset $\Gamma$ is written as $|\Gamma|$.

**3.8 Definition** The two-sided sequent calculus $\mathsf{K}_G$ is defined as follows:

a) Axioms:

1. $P, \Gamma \supset P, \Delta$
2. $\Gamma \supset \mathsf{true}, \Delta$
3. $\mathsf{false}, \Gamma \supset \Delta$

b) Logical rules:

1. Classical rules:

$$\frac{A, B, \Gamma \supset \Delta}{A \wedge B, \Gamma \supset \Delta}\ (\mathrm{l}\wedge) \qquad \frac{\Gamma \supset A, \Delta \quad \Gamma \supset B, \Delta}{\Gamma \supset A \wedge B, \Delta}\ (\mathrm{r}\wedge)$$

$$\frac{A, \Gamma \supset \Delta \quad B, \Gamma \supset \Delta}{A \vee B, \Gamma \supset \Delta}\ (\mathrm{l}\vee) \qquad \frac{\Gamma \supset A, B, \Delta}{\Gamma \supset A \vee B, \Delta}\ (\mathrm{r}\vee)$$

$$\frac{\Gamma \supset A, \Delta \quad B, \Gamma \supset \Delta}{A \to B, \Gamma \supset \Delta}\ (\mathrm{l} \to) \qquad \frac{A, \Gamma \supset B, \Delta}{\Gamma \supset A \to B, \Delta}\ (\mathrm{r} \to)$$

$$\frac{\Gamma \supset A, B, \Delta \quad A, B, \Gamma \supset \Delta}{A \leftrightarrow B, \Gamma \supset \Delta}\ (\mathrm{l} \leftrightarrow) \qquad \frac{A, \Gamma \supset B, \Delta \quad B, \Gamma \supset A, \Delta}{\Gamma \supset A \leftrightarrow B, \Delta}\ (\mathrm{r} \leftrightarrow)$$

$$\frac{\Gamma \supset A, \Delta}{\neg A, \Gamma \supset \Delta}\ (\mathrm{l}\neg) \qquad \frac{A, \Gamma \supset \Delta}{\Gamma \supset \neg A, \Delta}\ (\mathrm{r}\neg)$$

2. Modal rules:

$$\frac{A, \Gamma \supset \Delta}{\Diamond A, \Box\Gamma, \Sigma \supset \Diamond\Delta, \Pi}\ (\mathrm{l}\,\Diamond) \qquad \frac{\Gamma \supset A, \Delta}{\Box\Gamma, \Sigma \supset \Box A, \Diamond\Delta, \Pi}\ (\mathrm{r}\,\Box)$$

**3.9 Definition** The two-sided sequent calculus $\mathsf{KT}_G$ is defined as follows:

a) Axioms: as in Definition 3.8.

b) Logical rules:

1. Classical rules: as in Definition 3.8.
2. Modal rules:

$$\frac{A, \Gamma \supset \Delta}{\Diamond A, \Box\Gamma, \Sigma \supset \Diamond\Delta, \Pi}\ (\mathrm{l}\,\Diamond) \qquad \frac{\Gamma \supset A, \Diamond A, \Delta}{\Gamma \supset \Diamond A, \Delta}\ (\mathrm{r}\,\Diamond)$$

$$\frac{A, \Box A, \Gamma \supset \Delta}{\Box A, \Gamma \supset \Delta}\ (\mathrm{l}\,\Box) \qquad \frac{\Gamma \supset A, \Delta}{\Box\Gamma, \Sigma \supset \Box A, \Diamond\Delta, \Pi}\ (\mathrm{r}\,\Box)$$

**3.10 Definition** The two-sided sequent calculus $\mathsf{S4}_G$ is defined as follows:

a) Axioms: as in Definition 3.8.

b) Logical rules:

1. Classical rules: as in Definition 3.8.
2. Modal rules:

$$\frac{A, \Box\Gamma \supset \Diamond\Delta}{\Diamond A, \Box\Gamma, \Sigma \supset \Diamond\Delta, \Pi}\ (\mathrm{l}\,\Diamond) \qquad \frac{\Gamma \supset A, \Diamond A, \Delta}{\Gamma \supset \Diamond A, \Delta}\ (\mathrm{r}\,\Diamond)$$

$$\frac{A, \Box A, \Gamma \supset \Delta}{\Box A, \Gamma \supset \Delta}\ (\mathrm{l}\,\Box) \qquad \frac{\Box\Gamma \supset A, \Diamond\Delta}{\Box\Gamma, \Sigma \supset \Box A, \Diamond\Delta, \Pi}\ (\mathrm{r}\,\Box)$$

**3.11 Definition** Let $\mathsf{M}$ be one of $\mathsf{K}$, $\mathsf{KT}$, $\mathsf{S4}$. Then we write $\mathsf{M} \vdash \Gamma \supset \Delta$ if $\Gamma \supset \Delta$ is an axiom of $\mathsf{M}$, or if $\Gamma \supset \Delta$ can be deduced from axioms of $\mathsf{M}$ using classical and modal rules of $\mathsf{M}$. We write $\mathsf{M} \vdash A$ for $\mathsf{M} \vdash \emptyset \supset A$.

**Proof Search in the LWB 1.0**

For all three logics, the LWB 1.0 performs the proof search basically by backward applications of the above rules. For simplicity, refinements such as the use checking or the order of rule applications, have not been incorporated in the calculus. A termination criterion for $\mathsf{S4}_G$ (an efficient loop-checking) can be found in [42]. Of course, the implementation uses lists instead of multisets. However, the ordering of the lists is arbitrary and serves only to obtain a reproducible behavior.

**Backtracking:** The only rules that are not invertible, that is, rules where the conclusion may be provable while a premise is not, are $(\mathrm{l}\,\Diamond)$ and $(\mathrm{r}\,\Box)$. These rules enforce backtracking, if there are several possibilities to choose a $\Diamond A$ for the backward application of $(\mathrm{l}\,\Diamond)$ or a $\Box A$ for the backward application of $(\mathrm{r}\,\Box)$. The LWB 1.0 employs no heuristic for the choice of $\Diamond A$ and $\Box A$, respectively.

## 5.4 The Benchmark Formulae

### 5.4.1 Introduction

The starting point has been the development of a set of benchmark formulae which actually allows measuring the performance of different provers, and especially, to compare the sequential LWB 1.0 with the new distributed solution (see [41]).
The selection of formulae has the following properties:

1. Provable as well as unprovable formulae.

2. Formulae of various structures.

3. Scalability.

4. The result is known.

For each logic there are nine classes of provable formulae and nine classes of unprovable formulae. The formulae of a class can be parameterized by a natural number; the higher the number, the longer and the more complex the formula. This parameter will also be called the *level* of the formula. The scalability is essential to get formulae being hard enough.

The class names consist of a prefix indicating the logic, a middle part indicating the 'type' or origin, and a suffix **p** or **n** indicating whether the formulae of the class are provable or not. For instance, the formulae of the class **k_d4_p** are provable in the modal logic K. For each class of provable formulae there is a corresponding class of unprovable formulae. The definitions of the classes are not restated here; they can be found in [41] in full detail.

#### The Test Definition

In order to abstract from concrete timings (which depend on the machine type, the compiler, minor optimizations of the algorithm etc.), the benchmark test has been formulated in terms of the levels which the prover can reach for each class. More precisely, the result for a certain class is the highest level for which the prover can establish provability (or unprovability) within the time limit of 100 seconds. Having 18 classes per logic, the test result consists basically of 18 numbers.

#### Result Assessment

Of course, it can happen that some timings are around 100 seconds, in which case the result depends again for instance on the employed machine. So differences of $\pm 1$ level should either be ignored or carefully be examined. In short, the benchmark test is meant to detect *significant improvements* rather than the effect of detail optimizations. In a distributed environment, where we usually have different machines and where other factors, as for instance the network performance, influence the results, it is even more important to abstract from little differences. The time limit of 100 seconds permits to disregard optimizations, transformations, start-up times etc. in the order of magnitude of 3–4 seconds. This is certainly a benefit for the development of algorithms: they are not forced to care about such a fine granularity. The role of the hardware performance is discussed in the following section.

### 5.4.2 The Benchmark Formulae of the Modal Logic K

In the following section, we discuss some properties of the available test classes. The characterization of the formulae will help to assess the relevance of the results. See also the discussion of the formula classes in Sect. 5.6.4, after the first empirical results.

#### The Benchmark Result of the LWB 1.0

Table 5.1 lists the highest levels the LWB 1.0 could decide when run on a SPARCstation 5 with a 70MHz microSPARC II processor and 80 MB main memory. All results of the LWB 1.0 were computed on this machine.

**Influence of the Technical Equipment:** Experiments showed that the main memory has no importance (48 MB – 80 MB checked) for this test. When run on a machine which is $\sim 1.4$ times faster,[3] the LWB 1.0 reached only for three classes one level more: *k_path_n*, *k_poly_p* and *k_poly_n* (see also the corresponding diagrams in the following section). This shows that the test is quite well desensitized as regards medium changes of the test environment.

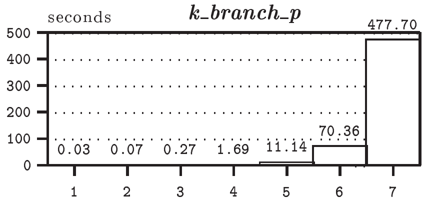
[3] Determined empirically, on a 110MHz microSPARC II.

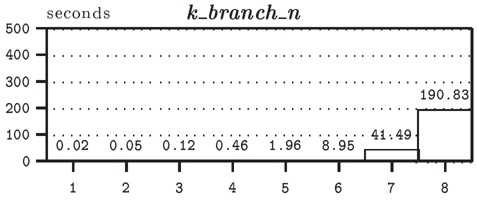
| class name | level | class name | level |
|---|---|---|---|
| *k_branch_p* | 6 | *k_branch_n* | 7 |
| *k_d4_p* | 8 | *k_d4_n* | 6 |
| *k_dum_p* | 13 | *k_dum_n* | 19 |
| *k_grz_p* | 7 | *k_grz_n* | 13 |
| *k_lin_p* | 11 | *k_lin_n* | 8 |
| *k_path_p* | 12 | *k_path_n* | 10 |
| *k_ph_p* | 4 | *k_ph_n* | 8 |
| *k_poly_p* | 8 | *k_poly_n* | 11 |
| *k_t4p_p* | 8 | *k_t4p_n* | 7 |

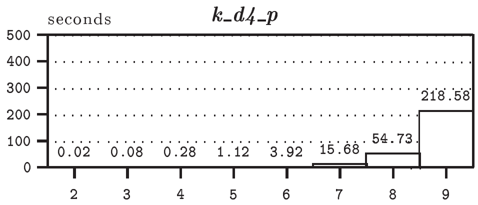
Table 5.1: Benchmark results of the LWB 1.0 for K.

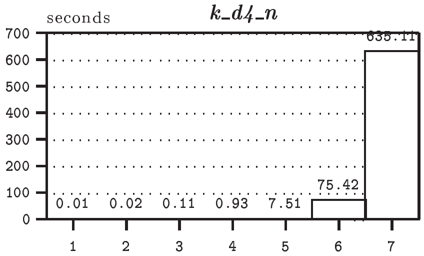
### Level – Time Diagrams

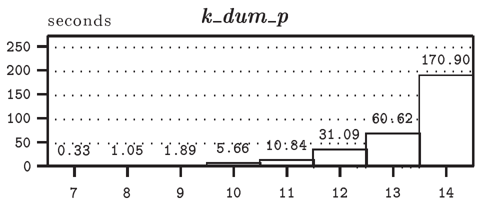
The following diagrams show the time consumption of the LWB 1.0 for (at most) 8 consecutive levels, ending with the first level which took more than 100 seconds.

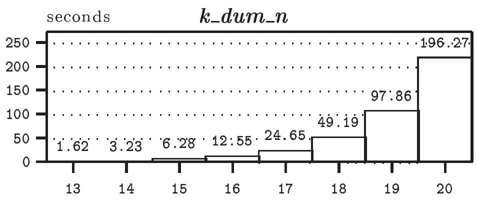


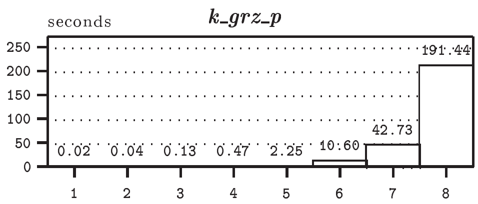



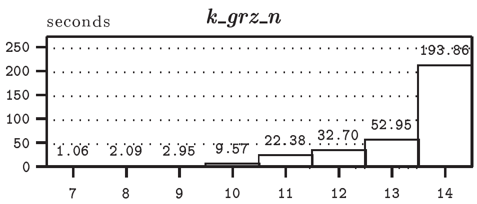



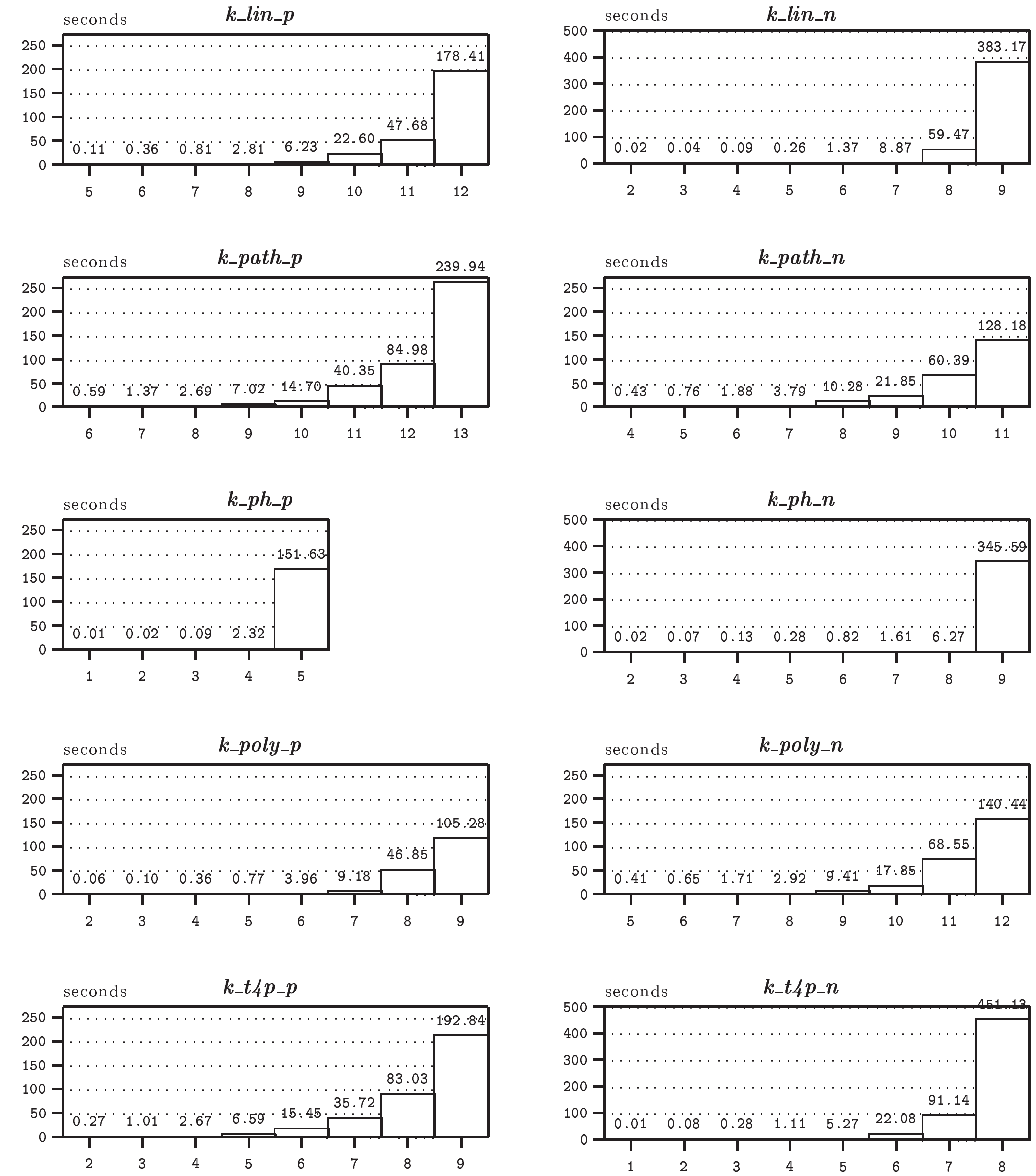






For all classes, the time consumption in the 100 seconds area grows exponentially. The factor from level to level is for most classes bigger than 2. A machine running at double speed would reach in no class 2 levels more.

## Level – Length

Table 5.2 lists the growth function of the formula length (see def 3.4) depending on the level. Let $C$ be a class name and $C(x)$ the formula of level $x$ of the class $C$. Then $L(C,x)$ is defined as $L(C,x) := \mathit{length}(C(x))$. Functions with a '$\sim$' prefix are least square approximations.

In short, the *lin* and *t4p* classes have linear growth, the *ph* classes have cubic growth, and the remaining classes have quadratic growth. Note that a linear increase

| class ($C$) | $L(C,x)$ | class ($C$) | $L(C,x)$ |
|---|---|---|---|
| *k_branch_p* | $44.5x^2 + 67.5x + 30$ | *k_branch_n* | $44.5x^2 + 66.5x + 27$ |
| *k_d4_p* | $3.5x^2 + 43.5x - 1$ | *k_d4_n* | $4x^2 + 69x - 1$ |
| *k_dum_p* | $\sim 0.5x^2 + 23.5x + 42$ | *k_dum_n* | $\sim 0.5x^2 + 27.5x + 44$ |
| *k_grz_p* | $\sim 4.5x^2 + 8x + 147$ | *k_grz_n* | $\sim 4.5x^2 + 9x + 145$ |
| *k_lin_p* | $\sim 80x + 15$ | *k_lin_n* | $\sim 184x - 219$ |
| *k_path_p* | $\sim 8x^2 + 101.5x - 79$ | *k_path_n* | $\sim 8x^2 + 120.0x + 46.3$ |
| *k_ph_p* | $2.5x^3 + 4x^2 + 1.5x + 1$ | *k_ph_n* | $2.5x^3 + 4x^2 + 2.5x + 1$ |
| *k_poly_p* | $\sim 4.5x^2 + 49.5x + 24$ | *k_poly_n* | $\sim 4.5x^2 + 46.5x + 8$ |
| *k_t4p_p* | $43x + 67$ | *k_t4p_n* | $86x + 16$ |

Table 5.2: Length of the $x^{\mathsf{th}}$ formula of a class.

of the length does not mean that the required time grows slower (see the previous diagrams).

**Level – Depth, Modal Depth**

The depth of a formula is defined as usual, while the modal depth is defined as the nesting depth of the modal operators $\Box$ and $\Diamond$ (cf. Definition 3.5 and 3.6). For most classes the growth of the depth and the modal depth is linear. The only exception as for the depth are the *ph* classes, which have a cubic growth. The classes with a constant modal depth are: *grz* (7), *lin* (3) and *ph* (2).

The modal depth indicates the maximum number of (r $\Box$) and (l $\Diamond$) applications on a branch of the proof search tree (in $\mathsf{K}_G$). After these applications, a classical sequent remains to be proved. In this sense, it measures how 'classical' a $\mathsf{K}$-formula is. Note, however, that at the points of the (r $\Box$) and (l $\Diamond$) applications, the branching factor may still be high. See, for instance, the *lin* classes in Table 5.7.

## 5.5 A K Calculus for Distribution

In this section we present a modification of the calculus $\mathsf{K}_G$ (definition 3.8). It will be the basis for the distributed, competitive proof search in $\mathsf{K}$. The adaption of the calculi $\mathsf{KT}_G$ and $\mathsf{S4}_G$ for distribution is analogous and is not discussed explicitly.

**5.1 Definition** A ternary function $f : \mathbb{N}^3 \to \mathbb{N}$ is called a *choice function* if

$$f(m,i,l) \in \{0 \ldots (l-1)\} \qquad \text{for all } (m,i,l) \text{ with } i < m.$$

The argument $m$ will later denote the number of machines (or processes), $i$ the id of the process $(0 \ldots (m-1))$, and $l$ the length of a list from which an element is chosen.

**5.2 Convention** Let $\Gamma$ be a multiset. Then we write $[\Gamma]$ for an arbitrary but fixed arrangement of the elements of $\Gamma$ in a list. $[\Gamma]_i$ is used for the $i^{\text{th}}$ element of $[\Gamma]$ $(i = 0 \ldots |\Gamma| - 1)$. Recall that $|\Phi|$ denotes the cardinality of the multiset $\Phi$.

**5.3 Definition** The calculus $\mathsf{K}_G^f(m,i)$, where $f$ is a choice function and $m, i \in \mathbb{N}$, $i < m$, is defined as follows :

a) Axioms: as in def 3.8.

b) Logical rules:

1. Classical rules: as in def 3.8.
2. Modal rules:

$$\frac{[\Phi]_j, \Gamma \supset \Delta}{\Diamond\Phi, \Box\Gamma, \Sigma \supset \Box\Psi, \Diamond\Delta, \Pi} \; j = f(m, i, |\Phi \cup \Psi|) < |\Phi| \quad (\mathrm{l}\,\Diamond)$$

$$\frac{\Diamond(\Phi \setminus [\Phi]_j), \Box\Gamma, \Sigma \supset \Box\Psi, \Diamond\Delta, \Pi}{\Diamond\Phi, \Box\Gamma, \Sigma \supset \Box\Psi, \Diamond\Delta, \Pi} \; j = f(m, i, |\Phi \cup \Psi|) < |\Phi| \quad (\mathrm{l}\,\Diamond_{\mathrm{bt}})$$

$$\frac{\Gamma \supset [\Psi]_j, \Delta}{\Diamond\Phi, \Box\Gamma, \Sigma \supset \Box\Psi, \Diamond\Delta, \Pi} \; 0 \le j = f(m, i, |\Phi \cup \Psi|) - |\Phi| \quad (\mathrm{r}\,\Box)$$

$$\frac{\Diamond\Phi, \Box\Gamma, \Sigma \supset \Box(\Psi \setminus [\Psi]_j), \Diamond\Delta, \Pi}{\Diamond\Phi, \Box\Gamma, \Sigma \supset \Box\Psi, \Diamond\Delta, \Pi} \; 0 \le j = f(m, i, |\Phi \cup \Psi|) - |\Phi| \quad (\mathrm{r}\,\Box_{\mathrm{bt}})$$

From the proof search point of view, $(\mathrm{l}\,\Diamond_{\mathrm{bt}})$ and $(\mathrm{r}\,\Box_{\mathrm{bt}})$ serve as 'backtracking rules': the $\Diamond$- or $\Box$-formula of $\Phi \cup \Psi$ that is selected by $f$ is either used for a backward application of $(\mathrm{l}\,\Diamond)$ or $(\mathrm{r}\,\Box)$, or it is removed from the sequent to enable the choice of another $\Diamond$- or $\Box$-formula (backtracking). They are not needed to preserve completeness (see the proof below).

**5.4 Theorem** *The calculus $\mathsf{K}_G^f(m,i)$ is correct and complete with respect to $\mathsf{K}_G$for any choice function $f$ and $m, i \in \mathbb{N}$, $i < m$.*

Proof: It is sufficient to show that the modal rules can be mutually simulated.

1. Correctness: the correctness of the $\mathsf{K}_G^f(m,i)$-rules $(\mathrm{l}\,\Diamond)$ and $(\mathrm{r}\,\Box)$ is trivial: simply choose the multisets $\Sigma$ and $\Pi$ in $(\mathrm{l}\,\Diamond)$ and $(\mathrm{r}\,\Box)$ of $\mathsf{K}_G$ accordingly. The $(\mathrm{l}\,\Diamond_{\mathrm{bt}})$ and $(\mathrm{r}\,\Box_{\mathrm{bt}})$ correspond to a left and right weakening, for which there is no rule in $\mathsf{K}_G$. Weakening, however, is built into the axioms of $\mathsf{K}_G$.

2. Completeness: to obtain a sequence derived by $(\mathrm{l}\,\Diamond)$ or $(\mathrm{r}\,\Box)$ of $\mathsf{K}_G$, apply the $(\mathrm{l}\,\Diamond)$ or $(\mathrm{r}\,\Box)$ of $\mathsf{K}_G^f(m,i)$ by setting $\Phi$ to $\{A\}$ and $\Psi$ to $\emptyset$ or vice versa for the $(\mathrm{r}\,\Box)$. The application is admissible since for any choice functions $f$ we have $f(m,i,l) = 0$ if $l = 1$ $(m, i \in \mathbb{N},\ i < m)$.

$\Box$

### 5.5.1 Choice Functions

**5.5 Definition** The function $f^{\mathrm{head}} : \mathbb{N}^3 \to \mathbb{N}$ is defined as

$$f^{\mathrm{head}}(m,i,l) := 0 \qquad \text{for all } m,i,l \in \mathbb{N}.$$

The function $f^{\mathrm{mod}} : \mathbb{N}^3 \to \mathbb{N}$ is defined as

$$f^{\mathrm{mod}}(m,i,l) := i \bmod l \qquad \text{for all } m,i,l \in \mathbb{N}.$$

The function $f^{\mathrm{rand}} : \mathbb{N}^3 \to \mathbb{N}$ is defined as

$$f^{\mathrm{rand}}(m,i,l) := \mathrm{rand}() \bmod l \qquad \text{for all } m,i,l \in \mathbb{N},$$

where $\mathrm{rand} : \mathbb{N} \to \mathbb{N}$ is an arbitrary random number generator.

$f^{\mathrm{head}}$, $f^{\mathrm{mod}}$, and $f^{\mathrm{rand}}$ are trivial choice functions. A more complex choice function is the following:

**5.6 Definition** The function $f^{\mathrm{sector}} : \mathbb{N}^3 \to \mathbb{N}$ is defined as

$$f^{\mathrm{sector}}(m,i,l) :=$$
$$\begin{cases} 0 & l \geq m \text{ and } i = 0 \\ l-1 & l \geq m \text{ and } i = m-1 \\ i(ldm+1) + ldm \operatorname{div} 2 & l \geq m \text{ and } 0 < i < lmm \\ lmm(ldm+1)+ & l \geq m \text{ and } lmm \leq i < (m-1) \\ \quad (i-lmm)ldm + (ldm-1) \operatorname{div} 2 & \\ i \operatorname{div}(mdl+1) & l < m \text{ and } i \leq mml(mdl+1) \\ mml+ & l < m \text{ and } i > mml(mdl+1) \\ \quad (i - mml(mdl+1)) \operatorname{div} mdl & \end{cases}$$

where $ldm = l \operatorname{div} m$ (list elements per machine), $lmm = l \bmod m$ (remainder), and $mdl = m \operatorname{div} l$ (machines per list element), $mml = m \bmod l$ (remainder).

Basically, if the remainders $lmm$ and $mml$ are 0 in the respective cases, we have $i\,ldm + (ldm - 1) \operatorname{div} 2$ for $l \geq m$ and $i \operatorname{div} mdl$ for $l < m$. The idea is to split a list of length $l$ into $m$ sectors, where $f^{\text{sector}}(m, i, l)$ returns the middle of the $i^{\text{th}}$ sector (if $l > m$). Exceptions: For the first sector, the first element is returned and for the last sector the last. The last two clauses describe the 'merging' of sectors when there are more sectors than list elements.

**Example ($f^{\text{sector}}$)**

Imagine a list $L$ of 11 natural numbers, $L := [0, 1, 2, \ldots, 9, 10]$. Having 4 machines $i = 0, 1, 2, 3$ that choose according to $f^{\text{sector}}(4, i, l)$ $(l = 11 \ldots 1)$, we obtain 4 different sequences. Define $L^i_j$ as: $L^i_0 := L$ and $L^i_{j+1}$ is the list $L^i_j$ without $s^i_j$, where $s^i_j$ is the $f^{\text{sector}}(4, i, 11 - j)^{\text{th}}$ element of $L^i_j$.

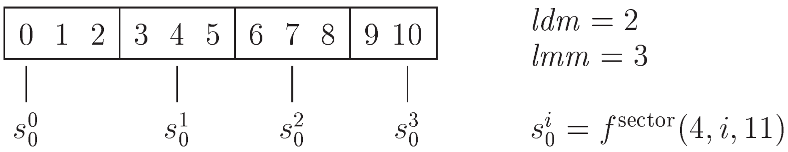



Figure 5.2: Dividing a list into sectors.

Table 5.3 shows the sequence of choices $s^i_j$ $(i = 0, 1, 2, 3,\ j = 0, \ldots, 10)$. Process 0

| $i \setminus j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 4 | 5 | 3 | 2 | 6 | 7 | 8 | 1 | 0 | 9 | 10 |
| 2 | 7 | 6 | 5 | 4 | 8 | 9 | 3 | 2 | 1 | 10 | 0 |
| 3 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Table 5.3: The different order of selection.

picks always the first and process 3 always the last element. Process 1 and 2 work from the middle outwards. Note that after three steps $(j = 0, 1, 2)$ each list item has been chosen by one of the 4 processes. Moreover, the order of the first two elements chosen by one process is inverted by at least one other process. This is important if one thinks of the numbers as proof tasks of different hardness. For example, if 5 represents a fast solution and 4 a hard task then process 1 will get stuck trying 4. However, provided that 7 and 6 are not hard as well, process 2 will find the fast solution in the third step.

## 5.6 Distributed Proof Search in the dLWB

### 5.6.1 OR- vs. AND-Parallelism

Proof search in one of the modal logics K, KT and S4 corresponds to a search in an AND-OR-graph. The AND-nodes correspond to backward applications of (l$\vee$), (r$\wedge$), (l $\leftrightarrow$) and (r $\leftrightarrow$) (see Definition 3.8), while the OR-nodes correspond to backward applications of the respective (l $\Diamond$) and (r $\Box$) rules.

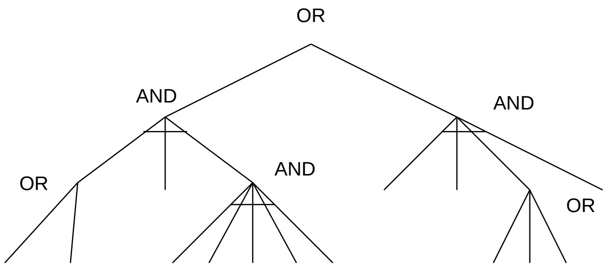



Figure 5.3: AND-OR-graph.

#### Duality between OR- and AND-nodes

There is a trivial duality between OR- and AND-nodes. For both types of nodes there exist cases in which all branches need to be done (opportunity for AND-parallelism) and cases in which the search can be aborted before all branches are done (opportunity for OR-parallelism).

- All branches must be examined:
  - AND-nodes: if all (but possibly the last) branches are successful.
  - OR-nodes: if all (but possibly the last) branches fail.
- The search can be aborted:
  - AND-nodes: if one branch fails ($\Rightarrow$ the node fails).
  - OR-nodes: if one branch succeeds ($\Rightarrow$ the node succeeds).

When doing proof search, a branch is 'successful' if the sequent of the corresponding premise is provable, and it 'fails' if the sequent is not provable. Since the calculi are complete and the search algorithms always terminate, the proof search will eventually detect whether or not a sequent is provable. In the following a short succeeding OR-branch will be called a 'fast solution'.

Since the test set contains provable as well as unprovable formulae, all four cases can occur. So depending on the result of the branches, an AND-node is actually an OR-node and vice versa. We could also say that we have two kinds of OR-nodes in proof search.

**Parallelization at AND-nodes:** Parallelization at AND-nodes is especially successful in cases in which there is a branch with a short failing search (left node in Fig. 5.4).


Figure 5.4: Parallelization at AND-nodes.

However, in the proof search of the considered logics, we can prune in certain cases the second of two successful branches (right node in Fig. 5.4). This technique is called 'use-checking' because it is checked whether the formula, which caused the branching, is actually used for the first subproof. To avoid a process doing useless work on a second branch that would later be pruned anyway, we abandon the option to parallelize at AND-nodes in the dLWB.

**Parallelization at OR-nodes:** Parallelization at OR-nodes is especially successful if there is a branch with a short successful search. At first sight, one might think that parallelization at OR-nodes will never lead to an exceptional speed-up if the original sequent is not provable (because all OR-branches are expected to fail). However, imagine an OR-node below an AND-node as in Fig. 5.5:

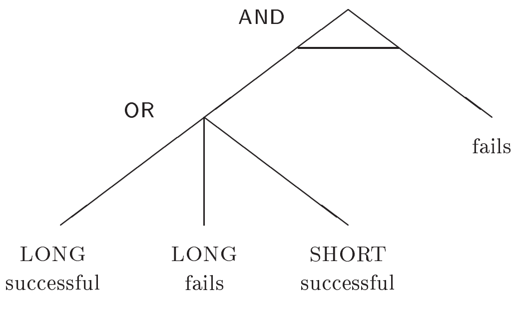



Figure 5.5: Success with parallelization at OR-nodes.

Since the overall search fails, at least one of the AND-branches must fail, but others may succeed (in the example the left branch with the OR-node), and the parallelization at OR-nodes may find a 'fast solution' on these branches. In fact, the dLWB also achieved significant speed-ups for classes of unprovable formulae.

### 5.6.2 Competition vs. Cooperation

#### Cooperation and Proof Search

The classical approach for distribution is certainly cooperative. The workload is split among the available processes. Father processes start up sons and collect the results. Very often, no further communication is needed and the size of the subtasks can be estimated. Moreover, most problems require only AND-parallelism because the result of all subtasks is relevant.

Proof search in non-classical logics, however, is more complex: the problem size is not known beforehand, the size and number of the subtasks cannot be estimated either, and OR-parallelism is fundamental, regardless of whether we parallelize at OR-nodes or AND-nodes, because in both cases the search may be aborted (see the section on duality above). In general, the search tree is highly irregular, and therefore a cooperative solution of such problems faces new difficulties:

- The machines should be able to help each other, to avoid situations in which only one machine works and the rest is idling. When a machine has finished a task, it should be assigned to a new one. So somehow a pool of unsolved subtasks must be available. The management of such a pool, however, can easily become a problem of its own, for instance, because the number of unsolved tasks might grow 'infinitely' during a search.
- The cost of packing, sending and unpacking is often disproportionate compared to the actual computation. To control this communication overhead, a fine-tuning is required leading to a very specialized design. In addition, bottlenecks may arise when more machines are added which create more and more tasks.
- Since OR-parallelism is involved, there must be a possibility to stop running tasks. For instance, when a task finds a 'fast solution', the other tasks of the above OR-node should be canceled. If a running task has created own subtasks, it can only be killed after these subtasks are stopped. So a parent-child relation must be maintained across the network (and the pool of unsolved tasks). Furthermore, better mechanisms than kill-and-restart of whole processes, which is simple but inefficient, must be found.

A successful cooperative approach will approximately attain a linear speed-up for problems without 'fast solution'. In view of the exponential growth of the time consumption (see Sect. 5.4.2), only few levels more can be expected.

#### Competition

By a competitive approach we mean a design in which the available machines run different strategies; the 'quickest' strategy (for a specific problem) 'wins'. There may be complete strategies which will always find the solution as well as partial strategies looking for fast solutions in a subset of the problem space. Such an approach has various advantages:

- It takes less effort to convert a sequential program into a distributed one. Moreover, new strategies can easily be added later.
- It is automatically safe in the sense that the loss of a machine does neither affect the termination nor the result.
- It can be more efficient to let fast machines 'overtake' slow ones, rather than to force them to communicate, especially if the slowness is due to a temporary high load.
- The competing strategies can still be adapted to cooperate voluntarily by exchanging partial results. This is also called 'cooperative competition'.

The drawback, however, is that the speed-up in cases in which no fast solutions exist is small. In addition, adding more computing power to successful cases will usually not improve the success significantly. Therefore, machines vs. speed-up diagrams are not the appropriate documentation of success (unlike for cooperative approaches).

## 5.6.3 The First Solution

The first successful solution was a distributed proof search according to different complete calculi, namely $\mathsf{K}_G^f(m,i)$ for some choice function $f$. For the present, the discussion is limited to the modal logic K; the best solution for K will later be carried over to KT and S4. The idea is to perform a 'virtual' parallelization at the OR-nodes of the search tree. The OR-branches are not really distributed, but they are searched in a different order (see Table 5.3). The situation for 5 machines is depicted in Fig. 5.6.

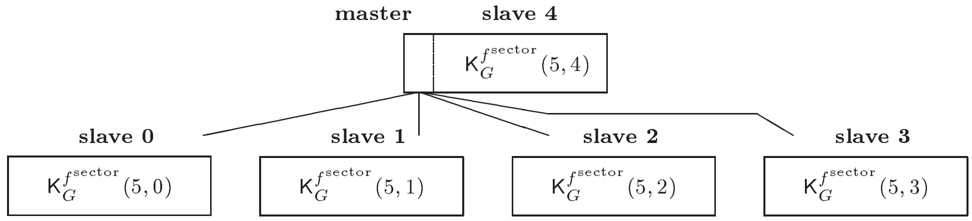



Figure 5.6: dLWB and $\mathsf{K}_G^{f^{\text{sector}}}(m,i)$.

A timeline of a distributed search in the dLWB with 3 machines is shown in Fig. 5.7. For clarity, the slave role of the master machine, when the master is being blocked, is omitted.

### Basic Comparisons

The first aim is of course to be significantly faster than the sequential LWB 1.0. Note that the LWB 1.0 performs the proof search basically according to $\mathsf{K}_G^{f^{\text{head}}}(m,i)$, which is equivalent to $\mathsf{K}_G^{f^{\text{sector}}}(m,0)$ and $\mathsf{K}_G^{f^{\text{mod}}}(m,0)$. All benchmark results of the

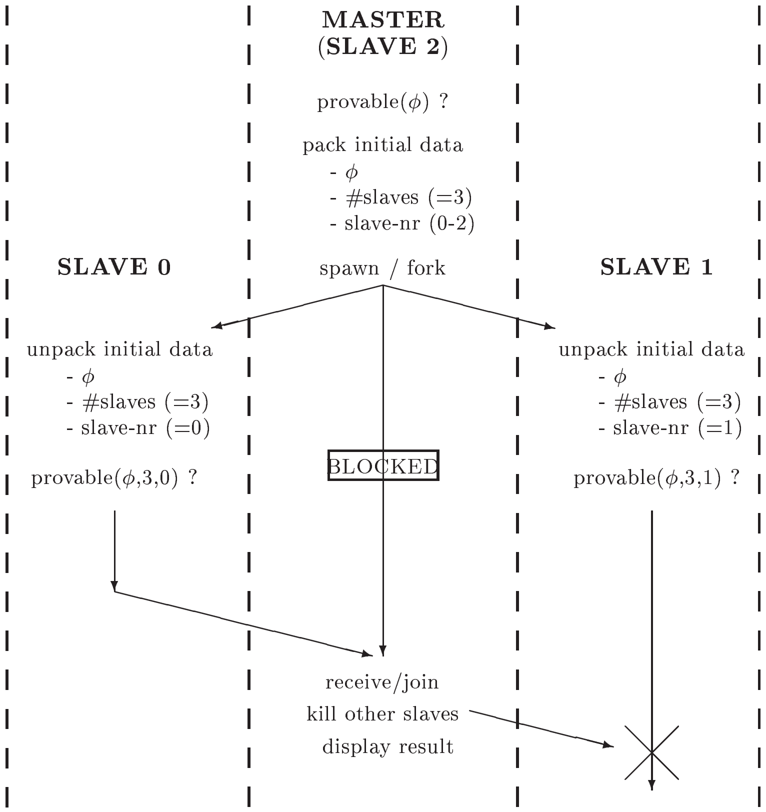



Figure 5.7: dLWB timeline (3 machines).

dLWB for K are relative to Table 5.1. Entries with '$\geq +9$' mean that 9 levels more are reached and possibly more could be reached. 9 levels more can be considered as a 'breakthrough'. Entries with '=' mean that the dLWB accomplishes the same number of levels.

An additional aim of a $\mathsf{K}_G^f(m,i)$-based distributed proof search is to be faster than the $\mathsf{K}_G^{f^{\mathrm{rand}}}(m,i)$-based proof search on the same number of machines $m$, that is, to be better than the random selection strategy.

All following results for 5 machines have been computed on 5 SPARCstation 5 (3 70MHz, 2 110MHz microSPARC II processors). For cases in which a higher level has only been reached because of the faster processor the respective entries in the tables are corrected.

### The Benchmark Result of the dLWB using $\mathsf{K}_G^{f^{\mathrm{rand}}}(5,i)$

| class name | level | class name | level |
|---|---|---|---|
| *k_branch_p* | −2 | *k_branch_n* | = |
| *k_d4_p* | −1 | *k_d4_n* | = |
| *k_dum_p* | +3 | *k_dum_n* | −1 |
| *k_grz_p* | +2 | *k_grz_n* | +2 |
| *k_lin_p* | +1 | *k_lin_n* | +1 |
| *k_path_p* | +2 | *k_path_n* | = |
| *k_ph_p* | = | *k_ph_n* | = |
| *k_poly_p* | +1 | *k_poly_n* | = |
| *k_t4p_p* | +1 | *k_t4p_n* | = |

Table 5.4: Benchmark results of the dLWB using $\mathsf{K}_G^{f^{\mathrm{rand}}}(5,i)$.

The standard C library function `rand()` is used as random number generator. To get different pseudo random number sequences, the $i^{\mathrm{th}}$ machine, which searches according to $\mathsf{K}_G^{f^{\mathrm{rand}}}(5,i)$, sets the $i$ as the seed value (function `srand()`).

Several tests were run with other random number sequences, namely with $5+i$, $10+i$, $15+i$ and $20+i$ as seed values for the $i^{\mathrm{th}}$ machine. The overall result did not change: 7–8 classes with higher levels and 2–3 classes with lower levels. The single results fluctuated of course. Among the best results were +8 for *k_dum_p* (seed $5+i$) and +7 for *k_path_p* (seed $20+i$). However, breakthroughs cannot be expected.

### The Benchmark Result of the dLWB using $\mathsf{K}_G^{f^{\mathrm{mod}}}(5,i)$

Using $f^{\mathrm{mod}}$ as the choice function, we obtain about the same improvement as with $\mathsf{K}_G^{f^{\mathrm{rand}}}(5,i)$. However, $f^{\mathrm{mod}}$ is certainly not an optimal strategy. When the machine $i$ has done the $i^{\mathrm{th}}$ OR-branch of an OR-node, it will continue with the $(i+1)^{\mathrm{st}}$ OR-branch, that is, the branch the machine $i+1$ has started with.

| class name | level | class name | level |
|---|---|---|---|
| *k_branch_p* | = | *k_branch_n* | = |
| *k_d4_p* | = | *k_d4_n* | = |
| *k_dum_p* | +5 | *k_dum_n* | −1 |
| *k_grz_p* | +3 | *k_grz_n* | +3 |
| *k_lin_p* | = | *k_lin_n* | +3 |
| *k_path_p* | ≥ 9 | *k_path_n* | = |
| *k_ph_p* | = | *k_ph_n* | = |
| *k_poly_p* | +1 | *k_poly_n* | = |
| *k_t4p_p* | +5 | *k_t4p_n* | = |

Table 5.5: Benchmark results of the dLWB using $\mathsf{K}_G^{f^{\text{mod}}}(5, i)$.

### The Benchmark Result of the dLWB using $\mathsf{K}_G^{f^{\text{sector}}}(5, i)$

| class name | level | class name | level |
|---|---|---|---|
| *k_branch_p* | = | *k_branch_n* | = |
| *k_d4_p* | +1 | *k_d4_n* | = |
| *k_dum_p* | ≥ +9 | *k_dum_n* | −1 |
| *k_grz_p* | ≥ +9 | *k_grz_n* | ≥ +9 |
| *k_lin_p* | ≥ +9 | *k_lin_n* | ≥ +9 |
| *k_path_p* | ≥ +9 | *k_path_n* | = |
| *k_ph_p* | = | *k_ph_n* | = |
| *k_poly_p* | +1 | *k_poly_n* | = |
| *k_t4p_p* | ≥ +9 | *k_t4p_n* | = |

Table 5.6: Benchmark results of the dLWB using $\mathsf{K}_G^{f^{\text{sector}}}(5, i)$.

$\mathsf{K}_G^{f^{\text{sector}}}(5, i)$ leads to breakthroughs in 7 classes! For instance from *k_grz_p*(7) (LWB 1.0, length 424) to *k_grz_p*(100) (length 45947), or from *k_path_p*(12) (LWB 1.0, length 2291) to *k_path_p*(50) (length 24996). The fast solutions are often found in the last 2 sectors, that is, by the search according to $\mathsf{K}_G^{f^{\text{sector}}}(5, 3)$ and $\mathsf{K}_G^{f^{\text{sector}}}(5, 4)$.

### Adding More Machines

It is not so surprising that doubling the number of machines (10 instead of 5) does not improve the results significantly. Using $\mathsf{K}_G^{f^{\text{rand}}}(10, i)$ basically returns the maximum of the results of $\mathsf{K}_G^{f^{\text{rand}}}(5, i)$ with seed $i$ and $\mathsf{K}_G^{f^{\text{rand}}}(5, i)$ with seed $5 + i$. Using $\mathsf{K}_G^{f^{\text{sector}}}(10, i)$ does not even lead to one level more. The dividing into twice as much sectors does not help. The following analysis of the formulae tries to give an explanation.

### 5.6.4 Discussion of the Formulae

A possibility to characterize a formula with respect to distribution is an investigation of the OR-nodes: for instance, when we employ 5 machines, the number of OR-nodes in a proof search tree is an interesting figure.

Of course, each calculus induces another search tree and so the relevance of such figures for other calculi is limited. Nevertheless, most calculi under consideration are closely related so that we can expect at least similar patterns in the statistics. Moreover, any proof search will visit large portions of the full search trees.

The statistics presented in Table 5.7 are based on the search trees of the LWB 1.0, that is, they correspond to $\mathsf{K}_G^{f^{\text{sector}}}(5,0)$. For each formula class, the search tree of the highest level reached by the LWB 1.0 is analyzed with respect to OR-nodes.

Table 5.7 lists the number of OR-nodes with respect to the number of branches. It further shows the maximum number of branches and the total number of OR-nodes.

- *k_branch_p* : This is a class of formulae in which the order in which OR-branches are treated does not seem to matter a lot. Neither $\mathsf{K}_G^{f^{\text{rand}}}(5,i)$ with different seeds helped, nor the other sectors of $\mathsf{K}_G^{f^{\text{sector}}}(5,i)$; selecting the first formula seems simply to be the best.
- *k_d4_p* : $\mathsf{K}_G^{f^{\text{sector}}}(5,i)$ got one additional level, but more sectors cannot help, because the maximum of OR-branches is 5.
- *k_grz_p*, *k_grz_n*, *k_lin_p*, *k_lin_n*, *k_path_p*: Here we have great improvements by $\mathsf{K}_G^{f^{\text{sector}}}(5,i)$ because there are many OR-nodes with many branches. Compared to the total number, the number of OR-nodes with more than 5 branches is relatively small.
- *k_ph_p*, *k_ph_n*, *k_poly_p*, *k_poly_p*: The small total number of OR-nodes unmasks these classes as 'basically classical'. The hard work is not in the modal part, so $\mathsf{K}_G^{f}(m,i)$ calculi will not help a lot.
- *k_t4p_p*: Is 'cracked' by $\mathsf{K}_G^{f^{\text{sector}}}(5,4)$. Even *k_t4p_p*$(1000)$ is no problem.
- *k_t4p_n*: More sectors cannot help because the maximum of OR-branches is 5.

The figures of Table 5.7 show that the classes of benchmark formulae have indeed different characteristics, which makes it hard for a prover to reach high levels for all of them. The figures explain to a certain degree that the use of more machines will not significantly improve proof search according to $\mathsf{K}_G^{f}(m,i)$ calculi for this set of formulae.

| formula | number of OR-nodes with given number of branches 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | max | total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *k_branch_p*(6) | 0 | 956 | 0 | 495 | 0 | 378 | 0 | 138 | 0 | 30 | 0 | 3 | | 12 | 2000 |
| *k_branch_n*(7) | 0 | 127 | | | | | | | | | | | | 2 | 127 |
| *k_d4_p*(8) | 221719 | 62918 | 87710 | 47873 | 1203 | | | | | | | | | 5 | 421423 |
| *k_d4_n*(6) | 158499 | 75450 | 44276 | 75389 | 55982 | 14535 | | | | | | | | 6 | 424131 |
| *k_dum_p*(13) | 170079 | 159931 | 3996 | | | | | | | | | | | 3 | 334006 |
| *k_dum_n*(19) | 149377 | 98533 | 14 | | | | | | | | | | | 3 | 247924 |
| *k_grz_p*(7) | 147457 | 77526 | 51752 | 7006 | 5183 | 2292 | 1100 | 291 | 39 | 2 | | | | 10 | 292648 |
| *k_grz_n*(13) | 100620 | 58324 | 39064 | 2190 | 6575 | 792 | 924 | 792 | 495 | 220 | 66 | 13 | 1 | 13 | 210076 |
| *k_lin_p*(11) | 11994 | 18736 | 19385 | 16139 | 13186 | 8223 | 3747 | 1153 | 247 | 39 | 7 | 3 | | 12 | 92859 |
| *k_lin_n*(8) | 11994 | 18736 | 19385 | 16139 | 13186 | 8223 | 3747 | 1153 | 247 | 39 | 7 | 3 | | 12 | 92859 |
| *k_path_n*(10) | 6 | 22392 | 7460 | 6 | | | | | | | | | | 4 | 29864 |
| *k_ph_p*(4) | 1 | | | | | | | | | | | | | 1 | 1 |
| *k_poly_p*(8) | 1 | 24 | 1 | | | | | | | | | | | 3 | 26 |
| *k_poly_n*(11) | 1091 | 66 | 1 | | | | | | | | | | | 3 | 1158 |
| *k_t4p_p*(8) | 375296 | 366029 | 119910 | 9851 | 94 | | | | | | | | | 5 | 871180 |
| *k_t4p_n*(7) | 656105 | 125417 | 60451 | 5017 | 13 | | | | | | | | | 5 | 847003 |

| formula | 1 | 2–21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | max | total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *k_path_p*(12) | 23061 | 0 | 1 | 18 | 141 | 630 | 1770 | 3260 | 3989 | 3259 | 1770 | 630 | 141 | 18 | 1 | 34 | 38689 |
| *k_ph_n*(8) | 1 | 0 | 0 | 0 | 1 | | | | | | | | | | | 24 | 2 |

Table 5.7: Number of OR-nodes in the proof search trees of the LWB 1.0.

### 5.6.5 Shaking the Formulae

The obvious objection to the good results of $\mathsf{K}_G^{f^{\text{sector}}}(m,i)$ (Table 5.6) is that they might depend on the specific structure of the formulae as it is given by the construction prescription in [41]. To investigate this dependence, a transformation function has been developed that rearranges a formula. Each transformation step must of course preserve equivalence.

**6.1 Proposition** *Let* $\mathsf{M}$ *be one of the calculi* $\mathsf{K}_G$, $\mathsf{KT}_G$, $\mathsf{S4}_G$, *and* $A, B, C \in \mathrm{Fml}$. *Then the following commutativity and associativity laws are provable in* $\mathsf{M}$.

$$\begin{array}{lcl}
\mathsf{M} & \vdash & (A \wedge B) \leftrightarrow (B \wedge A) \\
\mathsf{M} & \vdash & (A \vee B) \leftrightarrow (B \vee A) \\
\mathsf{M} & \vdash & (A \to (B \to C)) \leftrightarrow (B \to (A \to C)) \\
\mathsf{M} & \vdash & (A \leftrightarrow B) \leftrightarrow (B \leftrightarrow A) \\
\mathsf{M} & \vdash & (A \wedge (B \wedge C)) \leftrightarrow ((A \wedge B) \wedge C) \\
\mathsf{M} & \vdash & (A \vee (B \vee C)) \leftrightarrow ((A \vee B) \vee C) \\
\mathsf{M} & \vdash & (A \leftrightarrow (B \leftrightarrow C)) \leftrightarrow ((A \leftrightarrow B) \leftrightarrow C)
\end{array}$$

**6.2 Convention** FmlList is used for the set of all lists of formulae of Fml. $*$ : $\mathrm{FmlList} \times \mathrm{FmlList} \to \mathrm{FmlList}$ is used in infix notation for the concatenation of lists. $|L|$ denotes the length of the list $L$, $L_i$ the $i^{\text{th}}$ element of $L$, and $L - L_i$ the list $L$ without the element $L_i$.

Based on the above proposition we are going to define a recursive transformation function $T_{\text{sh}}$ that rearranges formulae. First, it collects the operands of 'neighboring' equal connectives ($c^{\cdots}$ functions, defined below), for which the associativity law holds. Then sh, which is some function which permutes a list of formulae, 'shakes' these operands (commutativity law). Finally, $T_{\text{sh}}$ rebuilds the formula in a left-associative manner ($r^{\cdots}$ functions).

**6.3 Definition** The conversion functions $r^{\wedge}, r^{\vee}, r^{\leftrightarrow}$ : FmlList $\to$ Fml, and $r^{\to}$ : FmlList $\times$ Fml $\to$ Fml are recursively defined as follows:

$$r^{\wedge}(L) := \begin{cases} \mathsf{true}, & L = [] \\ B, & L = [B] \\ r^{\wedge}(L') \wedge B, & L = L' * [B] \end{cases}$$

$$r^{\to}(L, C) := \begin{cases} C, & L = [] \\ B \to r^{\to}(L', C), & L = [B] * L' \end{cases}$$

$r^{\vee}$ and $r^{\leftrightarrow}$ are defined as $r^{\wedge}$ with $\wedge$ replaced by $\vee$ and $\leftrightarrow$, respectively. (The $r$ stands for 'rebuild'.)

**6.4 Definition** Let sh : FmlList $\to$ FmlList be an arbitrary function that permutes lists. Then the transformation function $T_{\text{sh}}$ : Fml $\to$ Fml and the conversion functions $c^{\wedge}, c^{\vee}, c^{\leftrightarrow}$ : Fml $\to$ FmlList, and $c^{\to}$ : Fml $\to$ FmlList $\times$ Fml are simultaneously defined:

$$c^{\wedge}(A) := \begin{cases} c^{\wedge}(A_1) * c^{\wedge}(A_1), & A = A_1 \wedge A_2 \\ [T_{\text{sh}}(A)], & \text{otherwise} \end{cases}$$

$$c^{\to}(A) := \begin{cases} ([T_{\text{sh}}(A_1)] * L,\ C), & A = A_1 \to A_2 \text{ and } c^{\to}(A_2) = (L, C) \\ ([],\ T_{\text{sh}}(A)), & \text{otherwise} \end{cases}$$

$$T_{\text{sh}}(A) := \begin{cases} r^{\wedge}(\text{sh}(c^{\wedge}(A))), & A = A_1 \wedge A_2 \\ r^{\vee}(\text{sh}(c^{\vee}(A))), & A = A_1 \vee A_2 \\ r^{\leftrightarrow}(\text{sh}(c^{\leftrightarrow}(A))), & A = A_1 \leftrightarrow A_2 \\ r^{\to}(\text{sh}(L), C), & A = A_1 \to (A_2 \to A_3) \text{ and } c^{\to}(A) = (L, C) \\ T_{\text{sh}}(A_1) \to T_{\text{sh}}(A_2), & A = A_1 \to A_2 \\ \neg T_{\text{sh}}(A_1), & A = \neg A_1 \\ \Box T_{\text{sh}}(A_1), & A = \Box A_1 \\ \Diamond T_{\text{sh}}(A_1), & A = \Diamond A_1 \\ A, & \text{otherwise} \end{cases}$$

$c^{\vee}$ and $c^{\leftrightarrow}$ are defined as $c^{\wedge}$ with $\wedge$ replaced by $\vee$ and $\leftrightarrow$, respectively. (The $c$ stands for 'collect'.)

**6.5 Theorem** *Let* $\mathsf{M}$ *be one of the calculi* $\mathsf{K}_G$, $\mathsf{KT}_G$, $\mathsf{S4}_G$, $A \in$ Fml *and* sh : FmlList $\to$ FmlList *an arbitrary function that permutes lists. Then the following equivalence is provable in* $M$*:*

$$\mathsf{M} \vdash A \leftrightarrow T_{\text{sh}}(A)$$

In the following we describe informally how we obtain a function $\text{rsh}_s$ ($s \in \mathbb{N}$) that 'randomly' permutes a list. First, we define a sequence of pseudo-random numbers:

**6.6 Definition** The sequence of pseudo-random numbers $\text{rd}_s : \mathbb{N} \to \mathbb{N}$ ( $s \in \mathbb{N}$ ) is defined as follows:

$$\text{rd}_s(n) := \begin{cases} s, & n = 0 \\ 29\,\text{rd}_s(n-1) \bmod 65537, & n > 0 \end{cases}$$

This is a simple form of a linear congruential pseudo-random number generator.

**6.7 Definition** A random permutation $p_{n,l} : \{1 \ldots l\} \to \{1 \ldots l\}$, $(l, n \in \mathbb{N})$,

$$\begin{pmatrix} 1 & 2 & \ldots & l \\ p_{n,l}(1) & p_{n,l}(2) & \ldots & p_{n,l}(l) \end{pmatrix}$$

is then defined as follows:

$$p_{n,l}(i) := \begin{cases} \mathrm{rd}_s(n) \bmod l, & i = 1 \\ \mathrm{rd}_s(n+j) \bmod l, & 1 < i \leq l, \text{ and for all } j' < j: \\ & (\mathrm{rd}_s(n+j') \bmod l) \in \{p_{n,l}(1), \ldots, p_{n,l}(i-1)\} \end{cases}$$

A sequence $S$ of random permutations $p_{n_0,l_0}, p_{n_1,l_1}, \ldots$ can then be obtained by defining $n_0$ to be 0 and $n_i$ to be $\min(\{n \mid n > n_{i-1}, \mathrm{rd}_s(n-1) \bmod l_{i-1} = p_{n_{i-1},l_{i-1}}(l_{i-1})\})$, that is, the random number generator continues with the first random number not 'used' so far.

Let $L_0, L_1, \ldots$ be the sequence of lists to be permuted by $T_{\mathrm{sh}}(A)$, where the order corresponds to a left-to-right traversal of $A$ (the enforcement of the order has not been defined in $T_{\mathrm{sh}}$ for better readability). Then $\mathrm{rsh}_s(L_i)$ is defined to be the list $L_i$ rearranged according to $p_{n_i,l_i}$, where $l_i$ is the length of $L_i$.

**Example:**

If $A$ is the formula

$$(p_{11} \wedge (p_{12} \wedge p_{13})) \vee \Big((p_{21} \wedge (p_{22} \wedge p_{23})) \vee (p_{31} \wedge (p_{32} \wedge p_{33}))\Big),$$

then

$$\begin{aligned} T_{\mathrm{rsh}_1}(A) &= \Big(((p_{13} \wedge p_{12}) \wedge p_{11}) \vee ((p_{31} \wedge p_{32}) \wedge p_{33})\Big) \vee ((p_{22} \wedge p_{23}) \wedge p_{21}) \\ T_{\mathrm{rsh}_7}(A) &= \Big(((p_{13} \wedge p_{12}) \wedge p_{11}) \vee ((p_{21} \wedge p_{22}) \wedge p_{23})\Big) \vee ((p_{33} \wedge p_{32}) \wedge p_{31}) \end{aligned}$$

If $A$ is the formula

$$(p_{11} \wedge (p_{12} \wedge p_{13})) \to \Big((p_{21} \wedge (p_{22} \wedge p_{23})) \to p_{31} \wedge (p_{32} \wedge p_{33})\Big),$$

then

$$T_{\mathrm{rsh}_1}(A) = ((p_{22} \wedge p_{23}) \wedge p_{21}) \to \Big(((p_{11} \wedge p_{12}) \wedge p_{13}) \to ((p_{33} \wedge p_{32}) \wedge p_{31})\Big).$$

### The Benchmark Result of the dLWB Using $\mathsf{K}_G^{f^{\mathrm{sector}}}(5,i)$ on Randomly Shaken Formulae

The following results are obtained when the transformation function $T_{\mathrm{rsh}_7}$ is applied (i.e. $\mathrm{rd}_7$ is used to generate the random permutations).

When the formulae are transformed, there is no longer a guarantee that the results are better or equal than those of the sequential LWB 1.0. In fact, for six classes the highest reached levels are lower, although 5 machines have been employed.

Compared to $\mathsf{K}_G^{f^{\mathrm{sector}}}(5,i)$ on unshaken formulae (Table 5.6), there are 7 classes with lower levels and 4 with higher levels. The guess, that a random shaking would

| class name | level | class name | level |
|---|---|---|---|
| *k_branch_p* | $-3$ | *k_branch_n* | $-1$ |
| *k_d4_p* | $-1$ | *k_d4_n* | $-1$ |
| *k_dum_p* | $\geq +9$ | ***k_dum_n*** | $\geq +9$ |
| *k_grz_p* | $\geq +9$ | *k_grz_n* | $\geq +9$ |
| *k_lin_p* | $\geq +9$ | *k_lin_n* | $+3$ |
| *k_path_p* | $-6$ | *k_path_n* | $-5$ |
| *k_ph_p* | $=$ | *k_ph_n* | $=$ |
| ***k_poly_p*** | $\geq +9$ | ***k_poly_n*** | $\geq +9$ |
| *k_t4p_p* | $\geq +9$ | ***k_t4p_n*** | $\geq +9$ |

Table 5.8: Benchmark results of the dLWB using $\mathsf{K}_G^{f^{\text{sector}}}(5, i)$ and $T_{\text{rsh}_7}$.

lead to a general deterioration, is refuted. On the contrary, the improvements for *k_dum_n*, *k_poly_p*, *k_poly_n* and *k_t4p_n* are proper breakthroughs!

It is quite surprising that the overall tendency does not depend on the way the formulae are shaken. Tests with the transformations $T_{\text{rsh}_3}$, $T_{\text{rsh}_{15}}$ and $T_{\text{rsh}_{17}}$ showed that in the 4 'successful' classes always (significantly) more levels were reached and in the 'failing' cases always less. It obviously matters *that* the formulae are shaken, rather than *how* they are shaken.

As regards the classical proof steps, the shaking corresponds to an alteration of rule applications, especially the invertible rules which branch in the proof search (backward applications of (r$\wedge$), (l$\vee$), (l $\rightarrow$), (r $\leftrightarrow$) and (l $\leftrightarrow$), see Definition 3.8). This accounts for the great improvements in the classes *k_poly_p* and *k_poly_n*, which are basically classical problems. So a part of the explanation is certainly that the shaking exploits to some degree the potential to speed up the proof search if the sequence of rule applications is changed. However, it does not account for the *general* improvement in some classes, except if we assume that the sequence of rule applications, as it is induced by the class definitions, is especially bad.

The random permutations lead sometimes to the odd situation that formulae of lower levels are harder to prove for the dLWB than formulae of higher levels. So the effect of faster machines is sometimes bigger compared to a monotonic increase of the time consumption, namely if a fast machine finds a proof of such an accidently hard formula.

### 5.6.6 Deletion of Modal Operators

A classical propositional formula can easily be acquired from a modal propositional formula by eliminating all modal operators $\Box$ and $\Diamond$.

**6.8 Definition** Let $A$ be a formula from Fml (Definition 3.2). Then $A^*$ is recursively defined as follows:

$$A^* := \begin{cases} B^*, & A = \Box B \text{ or } A = \Diamond B \\ \neg B^*, & A = \neg B \\ B^* \wedge C^*, & A = B \wedge C \\ B^* \vee C^*, & A = B \vee C \\ B^* \to C^*, & A = B \to C \\ B^* \leftrightarrow C^*, & A = B \leftrightarrow C \\ A, & \text{otherwise} \end{cases}$$

$A^*$ is obviously a classical propositional formula.

**6.9 Definition** The classical propositional calculus $\mathsf{CPC}_G$ is defined to be the calculus $\mathsf{K}$ (Definition 3.8) without modal rules.

**6.10 Theorem** *Let $\mathsf{M}$ be one of the calculi $\mathsf{K}_G$, $\mathsf{KT}_G$, $\mathsf{S4}_G$, $A \in \mathrm{Fml}$ and $\mathsf{CPC}_G$ the classical propositional calculus. Then we have the following implication:*

$$\mathsf{CPC}_G \nvdash A^* \quad \Rightarrow \quad \mathsf{M} \nvdash A$$

Proof: We prove by induction on the proof length that for all sequents $\Delta \supset \Gamma$

$$\mathsf{M} \vdash \Delta \supset \Gamma \quad \Rightarrow \quad \mathsf{CPC}_G \vdash \Delta^* \supset \Gamma^*,$$

where $\Delta^*$ and $\Gamma^*$ denote the multisets $\Gamma$ and $\Delta$ with all elements transformed according to the $^*$ transformation.

If $\Gamma \supset \Delta$ is an axiom of $\mathsf{M}$, for instance, $P, \Gamma_0 \supset P, \Delta_0$, then $P^*, \Gamma_0^* \supset P^*, \Delta_0^*$ is also an axiom of $\mathsf{CPC}_G$. For the induction step, we distinguish between the possible types of the last rule that has been applied. Assume that the last rule has been a classical rule, e.g. (r$\vee$). Then $\Gamma \supset \Delta = \Gamma_0 \supset A \vee B, \Delta_0$, and $\mathsf{M} \vdash \Gamma \supset A, B, \Delta$. By the induction hypothesis we have $\mathsf{CPC}_G \vdash \Gamma_0^* \supset A^*, B^*, \Delta_0^*$ and therefore $\mathsf{CPC}_G \vdash \Gamma_0^* \supset A^* \vee B^*, \Delta_0^*$ by a (r$\vee$) application of $\mathsf{CPC}_G$. The assertion is satisfied since $A^* \vee B^* = (A \vee B)^*$.

If the last rule has been a modal rule, e.g. (r$\Box$) of $\mathsf{K}$, then $\Gamma \supset \Delta = \Box\Gamma_0, \Sigma \supset \Box A, \Diamond\Delta_0, \Pi$, and $\mathsf{K} \vdash \Gamma_0 \supset A, \Delta_0$. The induction hypothesis yields $\mathsf{CPC}_G \vdash \Gamma_0^* \supset A^*, \Delta_0^*$, and therefore $\mathsf{CPC}_G$ proves also $\Gamma_0^*, \Sigma^* \supset A^*, \Delta_0^*, \Pi^*$ (add $\Sigma^*$ and $\Pi^*$ to all sequents in the proof of $\Gamma_0^* \supset A^*, \Delta_0^*$). The assertion is satisfied since $(\Box\Gamma_0)^* = \Gamma_0^*$, $(\Diamond\Delta_0)^* = \Delta_0^*$, and $(\Box A)^* = A^*$. The proofs for the other axioms and rules are analogous. □

The above theorem is now used to define 'speculative' variants of $\mathsf{K}_G$, $\mathsf{KT}_G$, and $\mathsf{S4}_G$. At certain stages of the proof search, the starred sequences are checked for classical provability; if the sequent is classically unprovable, then this branch of the search tree can be cut. Such a check is speculative since if the sequent *is* classically provable, then nothing is gained by the additional work. However, if the classical proof search fails, it fails usually much faster than the modal proof search because the sequent is simpler and no backtracking is needed. Moreover, the fastest classical prover at hand can be employed.

**6.11 Definition** The calculus $\mathsf{K}_G{}^{\mathrm{CPC}}$ is defined as follows:

a) Axioms: as in Definition 3.8.

b) Logical rules:

1. Classical rules: as in Definition 3.8.
2. Modal rules:

$$\frac{A^*, \Gamma^* \supset \Delta^* \quad A, \Gamma \supset \Delta}{\Diamond A, \Box\Gamma, \Sigma \supset \Diamond\Delta, \Pi}\,(\mathrm{l}\,\Diamond) \qquad \frac{\Gamma^* \supset A^*, \Delta^* \quad \Gamma \supset A, \Delta}{\Box\Gamma, \Sigma \supset \Box A, \Diamond\Delta, \Pi}\,(\mathrm{r}\,\Box)$$

Note that for proof search on starred sequents, $\mathsf{K}_G$ (and hence $\mathsf{K}_G{}^{\mathrm{CPC}}$) corresponds to $\mathsf{CPC}_G$. The calculi $\mathsf{KT}_G{}^{\mathrm{CPC}}$ and $\mathsf{S4}_G{}^{\mathrm{CPC}}$ are defined accordingly. The rules $(\mathrm{l}\,\Box)$ and $(\mathrm{r}\,\Diamond)$ stay the same.

It is obvious that $\mathsf{K}_G{}^{\mathrm{CPC}}$ is correct and complete with respect to $\mathsf{K}_G$. The correctness is trivial since we have only an additional premise in the modal rules. Conversely, we know by the previous theorem that if $\mathsf{K}_G$ proves a sequent then $\mathsf{CPC}_G$ proves the starred sequent, and therefore the additional premise in the modal rules represents no real restriction.

Note that it is sensible to check repeatedly for classical provability. Imagine the sequent $\Delta \supset \Gamma := (\emptyset \supset \neg P, \Box P, \Diamond\Delta_0, \Pi)$, with $\emptyset \supset P, \Delta_0$ being unprovable in $\mathsf{K}_G$. Of course, $\Delta^* \supset \Gamma^*$ is provable in $\mathsf{CPC}_G$. However, $\emptyset \supset P, \Delta_0^*$ might be unprovable in $\mathsf{CPC}_G$. Each backward application of a modal rule of $\mathsf{K}_G{}^{\mathrm{CPC}}$ involves usually a weakening; shorter sequents, however, are more likely to be classically unprovable.

### The Benchmark Result of the dLWB using $\mathsf{K}_G{}^{\mathrm{CPC}}$

The following results have been computed on a single machine. The results are again relative to Table 5.1.

As expected, the additional work slows the proving down in many cases. However, in most classes of unprovable formulae, an improvement is achieved. Moreover, even in two classes of provable formulae, higher levels are reached. For instance, in the class *k_dum_p* (at least) 180 levels (length 20472) instead of 13 levels (length 432) can be done; 184 classically unprovable sequents have been observed. Many hard and unprovable sequents close to the root of the search tree can obviously be sorted out. $\mathsf{K}_G{}^{\mathrm{CPC}}$ performs well for provable formulae if there are few successful branches among many failing branches at the backtrack points.

## 5.6.7 The Final Design of the dLWB

All benchmark formulae of a specific class have a certain common structure given by their parametric definition. However, also other formulae have usually a regularity

| class name | level | class name | level |
|---|---|---|---|
| *k_branch_p* | $-1$ | *k_branch_n* | $+1$ |
| *k_d4_p* | $=$ | *k_d4_n* | $+3$ |
| *k_dum_p* | $\geq +9$ | *k_dum_n* | $\geq +9$ |
| *k_grz_p* | $-1$ | *k_grz_n* | $-2$ |
| *k_lin_p* | $-1$ | *k_lin_n* | $\geq +9$ |
| *k_path_p* | $-2$ | *k_path_n* | $-1$ |
| *k_ph_p* | $=$ | *k_ph_n* | $+5$ |
| *k_poly_p* | $=$ | *k_poly_n* | $-3$ |
| *k_t4p_p* | $\geq +9$ | *k_t4p_n* | $\geq +9$ |

Table 5.9: Benchmark results of the dLWB using $\mathsf{K}_G{}^{\mathrm{CPC}}$.

resulting from the problem they describe. Very rarely, formulae are really random. So it can be assumed that in many cases the given structure generally eases the proof search, whereas in other cases it is a general handicap. It is therefore a natural approach for a distributed proof system to try to remove the handicap while still preserving the potential benefit of the structure. The solution chosen for the dLWB is to split the resources (machines) basically among two groups which compete with each other. One group proceeds according to $\mathsf{K}_G^{f^{\mathrm{sector}}}(m/2, i)$ on the formulae as they are, hoping for a structural benefit. The second group proceeds equally according to $\mathsf{K}_G^{f^{\mathrm{sector}}}(m/2, i)$ but on the shaken formulae, hoping that the shaking has dissolved a potential structural handicap. A single machine watching out for unprovable classical sequents (cf. the section above) completes the design. See Fig. 5.8.

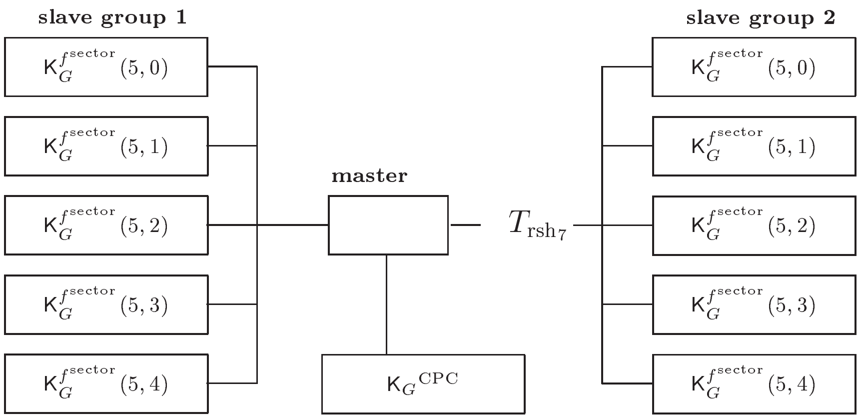



Figure 5.8: The final dLWB design.

The following results (Table 5.10) are obtained when the transformation function $T_{\mathrm{rsh}_7}$ is applied before the formulae are sent to the second slave group. Since 5 sectors turned out to be reasonable, the total number $m$ of employed machines is now 11.

In both slave groups there are 5 SPARCstation 5 (3 70MHz, 1 85MHz, 1 110MHz microSPARC II processors). The machine which proves according to $\mathsf{K}_G{}^{\text{CPC}}$ is a 110MHz SPARCstation 5. As before, the results are corrected if levels are reached only by faster machines.

| class name | level | class name | level |
|---|---|---|---|
| *k_branch_p* | = | *k_branch_n* | +1 |
| *k_d4_p* | +1 | *k_d4_n* | +3 |
| *k_dum_p* | $\geq +9$ | *k_dum_n* | $\geq +9$ |
| *k_grz_p* | $\geq +9$ | *k_grz_n* | $\geq +9$ |
| *k_lin_p* | $\geq +9$ | *k_lin_n* | $\geq +9$ |
| *k_path_p* | $\geq +9$ | *k_path_n* | = |
| *k_ph_p* | = | *k_ph_n* | +5 |
| *k_poly_p* | $\geq +9$ | *k_poly_n* | $\geq +9$ |
| *k_t4p_p* | $\geq +9$ | *k_t4p_n* | $\geq +9$ |

Table 5.10: Benchmark results of the dLWB using a combined approach.

**Summary (K):** The results are very convincing. In 11 out of 18 classes improvements are achieved which can be considered as breakthroughs. The transformation costs are not negligible (linear in the size of the formula) but the speed-up in the successful cases is much higher. Moreover, the transformation is done only once by the master, and it is carried out *after* the first slave group was started. That is, the transformation costs affect only the second group.

10 out of 11 machines have actually contributed to the results (that is, return as first for at least one level in a class). As pointed out in the previous sections, the addition of more machines will have no significant additional effect. The obvious idea to have a third group working on a differently but also randomly transformed input (e.g. $T_{\text{rsh}_3}$ instead of $T_{\text{rsh}_7}$) is not promising since the great improvements are in the same classes. However, transformations $T_{\text{sh}}$, in which sh rearranges lists of formulae according to some heuristic, could easily be incorporated in the same way if they are successful.

### 5.6.8 Moving to KT and S4

In this section the benchmark results of the dLWB for the modal logics KT and S4 are presented. The design stays the same as in Fig. 5.6 with $\mathsf{K}_G^{f^{\text{sector}}}(5,i)$ replaced by $\mathsf{KT}_G^{f^{\text{sector}}}(5,i)$ and $\mathsf{S4}_G^{f^{\text{sector}}}(5,i)$, respectively, and $\mathsf{K}_G{}^{\text{CPC}}$ replaced by $\mathsf{KT}_G{}^{\text{CPC}}$ and $\mathsf{S4}_G{}^{\text{CPC}}$, respectively.

An entry 'X' in a result table means that the corresponding class represented no problem for the LWB 1.0 (i.e. very high levels can be reached). The results are relative to the results of the LWB 1.0 (see [41]) for the absolute levels). The technical equipment is the same as in the corresponding investigation for K.

### The Benchmark Results for KT

Table 5.11 lists the results for the set of benchmark formulae for KT. The same combined approach as for K is used.

| class name | level | class name | level |
|---|---|---|---|
| *kt_45_p* | +2 | *kt_45_n* | +8 |
| *kt_branch_p* | +3 | *kt_branch_n* | = |
| *kt_dum_p* | $\geq +9$ | *kt_dum_n* | $\geq +9$ |
| *kt_grz_p* | $\geq +9$ | *kt_grz_n* | X |
| *kt_md_p* | +3 | *kt_md_n* | $\geq +9$ |
| *kt_path_p* | +8 | *kt_path_n* | +7 |
| *kt_ph_p* | = | *kt_ph_n* | = |
| *kt_poly_p* | = | *kt_poly_n* | = |
| *kt_t4p_p* | = | *kt_t4p_n* | $\geq +9$ |

Table 5.11: Benchmark results of the dLWB for KT using a combined approach.

In 11 out of 17 classes, a considerable improvement can be observed. Unfortunately, the shaking of the formulae does not help in a complementary way: the other methods reached higher levels in the 'successful' classes.

### The Benchmark Results for S4

Table 5.12 lists the results for the set of benchmark formulae for S4. The same combined approach as for K is used.

| class name | level | class name | level |
|---|---|---|---|
| *s4_45_p* | = | *s4_45_n* | +1 |
| *s4_branch_p* | +3 | *s4_branch_n* | +1 |
| *s4_grz_p* | $\geq +9$ | *s4_grz_n* | X |
| *s4_ipc_p* | $\geq +9$ | *s4_ipc_n* | $\geq +9$ |
| *s4_md_p* | +6 | *s4_md_n* | $\geq +9$ |
| *s4_path_p* | $\geq +9$ | *s4_path_n* | $\geq +9$ |
| *s4_ph_p* | = | *s4_ph_n* | = |
| *s4_s5_p* | $\geq +9$ | *s4_s5_n* | $\geq +9$ |
| *s4_t4p_p* | +5 | *s4_t4p_n* | +2 |

Table 5.12: Benchmark results of the dLWB for S4 using a combined approach.

In 12 out of 17 classes, a considerable improvement is achieved, in 8 classes we even have 'breakthroughs'. Again, the second slave group working on the shaken formulae performs rather poorly; only in one class (*s4_md_p*) the other methods do not reach higher levels. On the other hand, $\mathsf{S4}_G{}^{\mathrm{CPC}}$ is very successful: it even attained a breakthrough in a class of provable formulae (*s4_path_p*) in which the other methods take much longer.

**Summary (KT and S4):** The transfer of the distribution methods to KT and S4 has been straightforward and easy. Two methods work very well in these logics, too. The success of the third method is unfortunately not complementary. However, the overall results are very satisfying, especially if one takes into account that none of the differences between the logics has been exploited.

### 5.6.9 Conclusion

We have shown that a competitive approach can be very successful for distributed proof search in non-classical logics. Moreover, it seems very promising to let single machines apply 'speculative' methods which aim at certain 'weaknesses' of the formulae. Such methods need not be efficient in the general case, and they may even be incomplete. As long as they generate no communication, they can do no harm to the overall performance, but they can lead to an enormous speed-up in their special area.

The obvious advantage of a competitive approach are the modest requirements with regard to network performance and special software. In addition, the adaption and incorporation of sequential solutions requires little effort.

The inherent disadvantage of a purely competitive solution is that there will always be problem classes in which the progress is small. However, competition and cooperation do not exclude each other. Groups of cooperating processes may compete, or cooperating processes may have competing subtasks.

# Chapter 6

# A New One-Pass Tableau Calculus for PLTL

## 6.1 Introduction

Temporal logic can be viewed as a special kind of modal logic. Instead of describing that the truth of assertions may change in different worlds, the temporal operators or 'modalities' allow reasoning about different points in time.

Temporal logic has proved to be a useful formalism for reasoning about execution sequences of programs. It is especially appropriate to formulate and verify properties of nonterminating, reactive programs such as operating systems, communication protocols, traffic control systems, and concurrent programs (see for instance [12], [50], [53], [54]). A prominent variant is the propositional linear time logic PLTL of which the decision problem is known to be PSPACE-complete [61]. In most of the previous publications the decision algorithm itself has been presented as a 2-phase procedure:

1. A tableau procedure creating a graph.

2. A procedure checking whether the graph fulfills all eventuality formulae.

The second phase usually leads to an analysis of the strongly connected components (SCC) of the graph (see e.g. [64], [2]). Typical descriptions of this 2-phase method can be found in [66] and [36] where, in both cases, the second phase is not treated formally.

The tableau method presented in [47] is claimed to be incremental, where 'incremental' means that only reachable nodes are created (this is also true for [66] and [36]). However, it is essentially still a 2-phase procedure. The focus there is on providing a refined method for linear temporal logic with past time operators.

The verification problem can be formulated as a logical implication '*spec* $\rightarrow$ *prop*', where *spec* is the PLTL formula representing a specification and *prop* the formula representing a property to be verified. The essence of the problem is to show the validity of this implication in PLTL, which can be checked by any complete decision procedure for PLTL.

An alternative approach uses state-based methods (also referred to as 'model checking'). One possibility is to translate both the specification (e.g. of a protocol) and the negation of the property into labeled generalized Büchi automata, where the property automaton is also generated by a tableau-like procedure. A second phase then checks whether the language accepted by the synchronous product of the two automata is empty. Once again, in general, this involves an SCC analysis. In [31] it is claimed that the check for emptiness can be done 'on-the-fly' during the generation of the product: the tableau-like procedure builds the property graph in a depth-first manner choosing only successors which 'match' the current state of the protocol. Validity can also be checked using this method. However, it is not clear from the description whether the procedure remains 'on-the-fly' when there is no protocol to 'match'. In [16] it is shown how a generalized Büchi automaton can be transformed into a classical Büchi automaton for which the emptiness check reduces to a simple cycle detection scheme. So in the area of state-based methods similar attempts have been made to intermix the two phases and to avoid a standard SCC analysis.

Here we present a one-pass tableau calculus which checks locally, on-the-fly, for the fulfillment of eventuality formulae on a branch-by-branch basis. No second phase is required. It can also be used for an incremental depth-first search in which only reachable states are created. Derivations in this calculus result in (cyclic) tree-like structures rather than general graphs. Thus, the analysis of strongly connected components reduces to the detection of 'isolated subtrees', a task which is very simple and which can therefore be incorporated easily into the calculus. The new aspects basically consist of:

1. A branch-based loop check ensuring termination.
2. A part synthesizing the essential information gleaned from expanding the subtrees of a node.

The 2-phase methods require the creation of a fully expanded tableau, which is often exponential in the size of the initial formula. Since our method involves only one pass and is complete, we can stop as soon as a (counter-) model is detected, thus, (sometimes) avoiding a fully expanded tableau. A further advantage is that only one branch of the derivation tree needs to be considered at any stage. Therefore, the calculus $\mathsf{PLTL}_T$ is a natural analogue of the tableau and Gentzen-style sequent calculi for various modal logics, for instance K, KT and S4 (see e.g. [29], [35], [18]), in which derivations are also trees, in which it is always sufficient to consider one branch at any one time and in which a check for loops is sometimes required to guarantee termination (see e.g. [42]).

While the two phases of the previous methods are an obstacle for parallelization, the branch-by-branch treatment offers natural possibilities for concurrent search.

Of course, at the end, the resultant parts would need to be combined, but until then the processors could work independently on different subtrees without extra-communication.

There is of course a caveat. Since a naive derivation in $\mathsf{PLTL}_T$ essentially unfolds a graph into a tree, the run-time may be significantly higher, especially for examples in which the graphs have (relatively) few nodes and many edges. Clearly, the calculus must be applied in combination with suitable pruning and caching techniques. Such possibilities are discussed briefly at the end of the chapter. We will focus on the new definitions and the key lemmata and theorems. Simpler statements are formulated as propositions without proofs.

## 6.2 Syntax

In the following we deal with an extension $\mathcal{L}$ of the language for classical propositional logic which comprises the following symbols:

1. The countably many propositional variables $p_0, p_1, \ldots, p_i, \ldots$ for all natural numbers $i$.
2. The propositional constants $\mathsf{true}$ and $\mathsf{false}$.
3. The unary propositional connectives $\neg$ (not), $\mathsf{X}$ (neXt time), $\mathsf{F}$ (sometime in the future), $\mathsf{G}$ (generally, always in the future).
4. The binary propositional connectives $\wedge$ (and), $\vee$ (or), $\mathcal{U}$ (until) and $\mathcal{B}$ (before).

As auxiliary symbols we have parentheses and commas. The set of propositional variables is denoted by Var. Propositional variables are also called *positive literals*; if $P$ is a propositional variable then $\neg P$ is a *negative literal*. As metavariable for positive literals we use $P$ and as metavariable for literals $M$, possibly with subscripts.

**2.1 Definition** The formulae of $\mathcal{L}$ are inductively defined as follows:

1. All propositional variables and propositional constants of $\mathcal{L}$ are formulae of $\mathcal{L}$.
2. If $A$ is a formula of $\mathcal{L}$, then $(\neg A)$, $(\mathsf{X}\,A)$, $(\mathsf{F}\,A)$, and $(\mathsf{G}\,A)$ are formulae of $\mathcal{L}$.
3. If $A$ and $C$ are formulae of $\mathcal{L}$, then $(A \wedge C)$, $(A \vee C)$, $(A\,\mathcal{U}\,C)$, $(A\,\mathcal{B}\,C)$ are formulae of $\mathcal{L}$.

Implication and equivalence are defined as usual:

$$(A \to C) := (\neg A \vee C) \quad \text{and} \quad (A \leftrightarrow C) := ((A \to C) \wedge (C \to A)).$$

We denote the set of all formulae of $\mathcal{L}$ by Fml. A subset of Fml is also called a *theory*. As metavariables for propositional variables we use $P, Q$, and as metavariables for formulae we use $A, C, D$, possibly with subscripts.

Formulae which contain the symbol $\neg$ only immediately before positive literals are called formulae in *negation normal form*. The following function *nnf* yields for any formula an equivalent formula in negation normal form.

**2.2 Definition** The function $nnf : \mathrm{Fml} \to \mathrm{Fml}$ is inductively defined as follows:

| $A$ | $nnf(A)$ | $nnf(\neg A)$ |
|---|---|---|
| $\mathsf{true}$ | $\mathsf{true}$ | $\mathsf{false}$ |
| $\mathsf{false}$ | $\mathsf{false}$ | $\mathsf{true}$ |
| $P$ | $P$ | $\neg P$ |
| $\neg C$ | $nnf(\neg C)$ | $nnf(C)$ |
| $C \wedge D$ | $nnf(C) \wedge nnf(D)$ | $nnf(\neg C) \vee nnf(\neg D)$ |
| $C \vee D$ | $nnf(C) \vee nnf(D)$ | $nnf(\neg C) \wedge nnf(\neg D)$ |
| $\mathsf{X}\,C$ | $\mathsf{X}\,(nnf(C))$ | $\mathsf{X}\,(nnf(\neg C))$ |
| $\mathsf{G}\,C$ | $\mathsf{G}\,(nnf(C))$ | $\mathsf{F}\,(nnf(\neg C))$ |
| $\mathsf{F}\,C$ | $\mathsf{F}\,(nnf(C))$ | $\mathsf{G}\,(nnf(\neg C))$ |
| $C\,\mathcal{U}\,D$ | $nnf(C)\,\mathcal{U}\,nnf(D)$ | $nnf(\neg C)\,\mathcal{B}\,nnf(D)$ |
| $C\,\mathcal{B}\,D$ | $nnf(C)\,\mathcal{B}\,nnf(D)$ | $nnf(\neg C)\,\mathcal{U}\,nnf(D)$ |

The *complement* $\overline{A}$ of a formula $A$ is then defined as $\overline{A} := nnf(\neg A)$.

**2.3 Proposition** *For any formula $A$ we have that $nnf(A)$ and the complement $\overline{A}$ are in negation normal form.*

**2.4 Definition** The *literal length* $litlen(A)$ of a formula $A$ is inductively defined as follows.

1. $litlen(A) := 1$ if $A$ is a propositional variable or a constant.
2. $litlen(A) := litlen(C)$ if $A$ has the form $\neg C$.
3. $litlen(A) := litlen(C) + 1$ if $A$ has the form $\mathsf{X}\,C$, $\mathsf{G}\,C$, or $\mathsf{F}\,C$.
4. $litlen(A) := litlen(C) + litlen(D) + 1$ if $A$ has the form $C \wedge D$, $C \vee D$, $C\,\mathcal{B}\,D$ or $C\,\mathcal{U}\,D$.

**2.5 Proposition** *For all formulae $A$ we have $litlen(A) = litlen(\overline{A})$.*

In the following we usually speak of formulae when we have formulae of $\mathcal{L}$ in mind. In order to increase readability it makes sense to avoid too many parentheses. As a convention we omit outer parentheses and define the unary connectives to take precedence over all binary connectives. In addition, we simply write

$$\begin{array}{lll} A_0 \vee \ldots \vee A_{n-1} & \text{for} & (\ldots(A_0 \vee A_1) \vee \ldots \vee A_{n-1}), \\ A_0 \wedge \ldots \wedge A_{n-1} & \text{for} & (\ldots(A_0 \wedge A_1) \wedge \ldots \wedge A_{n-1}). \end{array}$$

**2.6 Example** We list some examples of formulae:

(i) $((\mathsf{X}\, p_1)\, \mathcal{B}\, \mathsf{true})$,

(ii) $(p_0)$,

(iii) $((\mathsf{F}\, (p_7\, \mathcal{U}\, p_1)) \wedge (p_0\, \mathcal{B}\, (\neg(\mathsf{X}\, p_1))))$,

(iv) $(\mathsf{G}\, (((p_0 \vee p_1) \vee p_2) \vee p_3))$.

Using our conventions, these formulae can also be written as:

(i) $\mathsf{X}\, p_1\, \mathcal{B}\, \mathsf{true}$,

(ii) $p_0$,

(iii) $\mathsf{F}\, (p_7\, \mathcal{U}\, p_1) \wedge (p_0\, \mathcal{B}\, \neg\mathsf{X}\, p_1)$,

(iv) $\mathsf{G}\, (p_0 \vee p_1 \vee p_2 \vee p_3)$.

## 6.3 Semantics

**3.1 Definition** A PLTL-*model* is a pair $\langle S, L\rangle$, where:

1. $S$ is an infinite sequence of states $(s_i)_{i\in\mathbb{N}} = s_0\, s_1 \ldots$ .
2. $L : S \to \mathrm{Pow}(\mathrm{Var})$ is a function which assigns to each state a set of propositional variables. $L$ is called a 'labeling'.

The states $s_i$ of a PLTL-model are arbitrary elements. We represent models as directed linear graphs. The vertices of the graph correspond to the states in $S$, and the arrows correspond to the successor relation induced by the sequence. The variables we list in a state $s_i$ of a model $\langle S, L\rangle$ are exactly the variables of the set $L(s_i)$. We draw a loop from state $s_{i+l}$ to state $s_i$ and omit all further states if the labeling $L$ repeats itself, that is, if $p_j \in L(s_{i+n}) \Leftrightarrow p_j \in L(s_{i+(l+1)+n})$ for all natural numbers $j$ and $n$. We occasionally add negated variables to a state to emphasize that these variables may not be true at that state.

**3.2 Example** Let $\mathcal{M}$ be the PLTL-model $\langle S, L\rangle$, where:

- $S = s_0\, s_1\, s_2\, s_3\, \ldots$ .
- $L(s_0) = \{p_0\}$, $L(s_{1+2n}) = \{p_1, p_2\}$, $L(s_{2+2n}) = \{p_1\}$ for all natural numbers $n$.

The upper part of Fig. 6.1 shows a 'prefix' of the above model. Using the loop convention the 'full' model can be displayed (lower part).

**3.3 Definition** If $\mathcal{M} = \langle S, L\rangle$ is a PLTL-model and $s_i \in S$, then the value $\mathcal{M}(s_i, A)$ of a formula $A$ in state $s_i$ is inductively defined as follows:

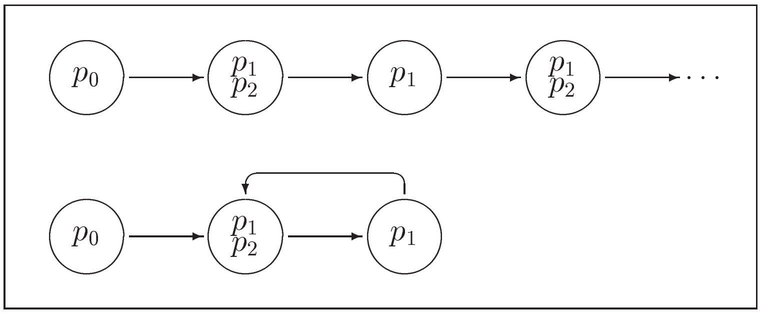



Figure 6.1: Drawings of a PLTL-model, with and without a final loop.

1. $\mathcal{M}(s_i, P) := \mathsf{t}$ if $P \in L(s_i)$ and $\mathcal{M}(s_i, P) := \mathsf{f}$ if $P \notin L(s_i)$.

2. $\mathcal{M}(s_i, \mathsf{true}) := \mathsf{t}$ and $\mathcal{M}(s_i, \mathsf{false}) := \mathsf{f}$.

3. $\mathcal{M}(s_i, \neg A) := \begin{cases} \mathsf{t} & \text{if } \mathcal{M}(s_i, A) = \mathsf{f}, \\ \mathsf{f} & \text{otherwise.} \end{cases}$

4. $\mathcal{M}(s_i, A \wedge C) := \begin{cases} \mathsf{t} & \text{if } \mathcal{M}(s_i, A) = \mathsf{t} \text{ and } \mathcal{M}(s_i, C) = \mathsf{t}, \\ \mathsf{f} & \text{otherwise.} \end{cases}$

5. $\mathcal{M}(s_i, A \vee C) := \begin{cases} \mathsf{t} & \text{if } \mathcal{M}(s_i, A) = \mathsf{t} \text{ or } \mathcal{M}(s_i, C) = \mathsf{t}, \\ \mathsf{f} & \text{otherwise.} \end{cases}$

6. $\mathcal{M}(s_i, \mathsf{X}\, A) := \begin{cases} \mathsf{t} & \text{if } \mathcal{M}(s_{i+1}, A) = \mathsf{t}, \\ \mathsf{f} & \text{otherwise.} \end{cases}$

7. $\mathcal{M}(s_i, \mathsf{G}\, A) := \begin{cases} \mathsf{t} & \text{if } \mathcal{M}(s_{i+j}, A) = \mathsf{t} \text{ for all } j \geq 0, \\ \mathsf{f} & \text{otherwise.} \end{cases}$

8. $\mathcal{M}(s_i, \mathsf{F}\, A) := \begin{cases} \mathsf{t} & \text{if there exists a } j \geq 0 \text{ such that } \mathcal{M}(s_{i+j}, A) = \mathsf{t}, \\ \mathsf{f} & \text{otherwise.} \end{cases}$

9. $\mathcal{M}(s_i, A\, \mathcal{U}\, C) := \begin{cases} \mathsf{t} & \text{if there exists a } j \geq 0 \text{ such that } \mathcal{M}(s_{i+j}, C) = \mathsf{t} \text{ and} \\ & \mathcal{M}(s_{i+k}, A) = \mathsf{t} \text{ for all } 0 \leq k < j, \\ \mathsf{f} & \text{otherwise.} \end{cases}$

10. $\mathcal{M}(s_i, A\, \mathcal{B}\, C) := \begin{cases} \mathsf{t} & \text{if for all } j \geq 0 \text{ with } \mathcal{M}(s_{i+j}, C) = \mathsf{t} \\ & \text{there exists a } 0 \leq k < j \text{ with } \mathcal{M}(s_{i+k}, A) = \mathsf{t}, \\ \mathsf{f} & \text{otherwise.} \end{cases}$

In the sequel we abbreviate $\mathcal{M}(s_i, A) = \mathsf{t}$ to $\mathcal{M}, s_i \Vdash A$ and we abbreviate $\mathcal{M}(s_i, A) = \mathsf{f}$ to $\mathcal{M}, s_i \nVdash A$. If $\mathcal{M}, s_i \Vdash A$ for all $s_i \in S$ then we write $\mathcal{M} \Vdash A$.

Observe that the boolean connectives are treated as in the case of classical propositional logic, although relativized to each state. This means that the value of a boolean connected formula in state $s_i$ can be computed from the values of its components in $s_i$. This is in sharp contrast to a formula connected with a temporal operator in which the value in a state $s_i$ depends on the values of its components in the future. Figure 6.2 shows the corresponding drawing.

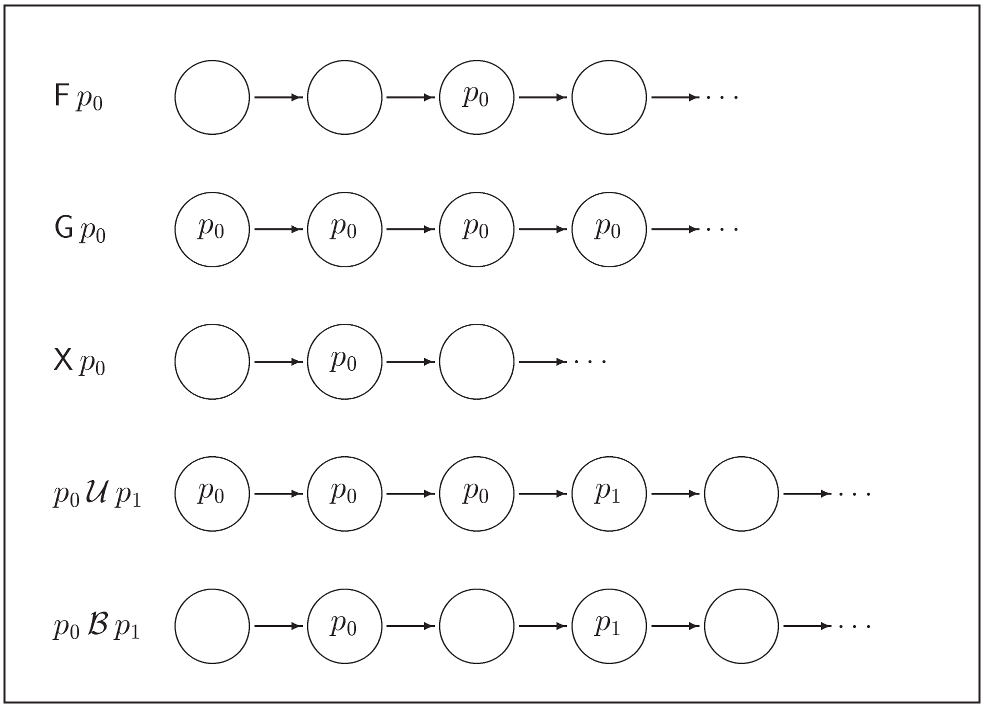



Figure 6.2: Drawings of PLTL-models of simple temporal formulae.

Note that in a PLTL-model $\mathcal{M} = \langle S, L \rangle$ truth need not to be preserved in the future, that is, if $\mathcal{M}, s_i \Vdash A$, we may have $\mathcal{M}, s_j \nVdash A$ for some $j > i$ and vice versa.

Using the definition of PLTL-models we can now define PLTL-satisfiability and PLTL-validity for formulae in the 'usual' way.

**3.4 Definition**

1. A formula $A$ is PLTL-*satisfiable* if and only if there exists a PLTL-model $\mathcal{M} = \langle S, L \rangle$ and a state $s_i \in S$ such that $\mathcal{M}, s_i \Vdash A$.

2. A formula $A$ is PLTL-*valid* if and only if $\mathcal{M} \Vdash A$ for all PLTL-models $\mathcal{M} = \langle S, L \rangle$. Then we write PLTL $\models A$.

**3.5 Definition**

1. A PLTL-model $\mathcal{M} = \langle S, L \rangle$ PLTL-satisfies a theory $Th$ if $\mathcal{M} \Vdash A$ for all $A \in Th$.

2. A formula $A$ is a PLTL-consequence of the theory $Th$ if, for all PLTL-models $\mathcal{M} = \langle S, L \rangle$ that PLTL-satisfy $Th$, we have $\mathcal{M} \Vdash A$. Then we write PLTL $+ Th \models A$.

In the next three propositions we collect a series of PLTL-valid formulae. In each case the validity can be checked by trivial calculations with PLTL-models.

**3.6 Proposition** *The following* PLTL*-valid equivalences can be used to construct equivalent* PLTL *formulae in negation normal form. Of course, the classical equivalences that allow us to push the negation inwards also hold.*

- $\mathsf{PLTL} \models \neg \mathsf{X}\, A \leftrightarrow \mathsf{X}\, \neg A$
- $\mathsf{PLTL} \models \neg \mathsf{F}\, A \leftrightarrow \mathsf{G}\, \neg A$
- $\mathsf{PLTL} \models \neg \mathsf{G}\, A \leftrightarrow \mathsf{F}\, \neg A$
- $\mathsf{PLTL} \models \neg(A \,\mathcal{U}\, C) \leftrightarrow (\neg A \,\mathcal{B}\, C)$
- $\mathsf{PLTL} \models \neg(A \,\mathcal{B}\, C) \leftrightarrow (\neg A \,\mathcal{U}\, C)$
- $\mathsf{PLTL} \models \mathsf{F}\, A \leftrightarrow (\mathsf{true}\,\mathcal{U}\, A)$

Note especially the last equivalence. Since any classical connective can be expressed in terms of $\neg$ and $\vee$ and $\mathsf{F}$ and $\mathsf{G}$ can be expressed in terms of $\mathcal{U}$, we immediately have:

**3.7 Corollary** *For any* PLTL *formula there exists an equivalent* PLTL *formula in which only the connectives* $\neg$, $\vee$, $\mathsf{X}$ *and* $\mathcal{U}$ *occur.*

**3.8 Proposition** *For any formula* $A$ *we have* $\mathsf{PLTL} \models A \leftrightarrow \mathit{nnf}(A)$.

**3.9 Proposition** *The following* PLTL*-valid formulae show distributivity laws of the temporal operators.*

- $\mathsf{PLTL} \models \mathsf{X}\,(A \vee C) \leftrightarrow (\mathsf{X}\, A \vee \mathsf{X}\, C)$
- $\mathsf{PLTL} \models \mathsf{X}\,(A \wedge C) \leftrightarrow (\mathsf{X}\, A \wedge \mathsf{X}\, C)$
- $\mathsf{PLTL} \models \mathsf{X}\,(A \,\mathcal{U}\, C) \leftrightarrow ((\mathsf{X}\, A)\,\mathcal{U}\,(\mathsf{X}\, C))$
- $\mathsf{PLTL} \models \mathsf{X}\,(A \,\mathcal{B}\, C) \leftrightarrow ((\mathsf{X}\, A)\,\mathcal{B}\,(\mathsf{X}\, C))$
- $\mathsf{PLTL} \models ((A \wedge C)\,\mathcal{U}\, D) \leftrightarrow ((A \,\mathcal{U}\, D) \wedge (C \,\mathcal{U}\, D))$
- $\mathsf{PLTL} \models (A \,\mathcal{U}\,(C \vee D)) \leftrightarrow ((A \,\mathcal{U}\, C) \vee (A \,\mathcal{U}\, D))$
- $\mathsf{PLTL} \models ((A \vee C)\,\mathcal{B}\, D) \leftrightarrow ((A \,\mathcal{B}\, D) \vee (C \,\mathcal{B}\, D))$
- $\mathsf{PLTL} \models (A \,\mathcal{B}\,(C \vee D)) \leftrightarrow ((A \,\mathcal{B}\, C) \wedge (A \,\mathcal{B}\, D))$

**3.10 Proposition** *The following formulae concerning* true *and* false *are* PLTL*-valid. Only the cases for* $\mathsf{X}$ *and* $\mathcal{U}$ *are stated.*

- $\mathsf{PLTL} \models \mathsf{X}\,\mathsf{true} \leftrightarrow \mathsf{true}$
- $\mathsf{PLTL} \models \mathsf{X}\,\mathsf{false} \leftrightarrow \mathsf{false}$
- $\mathsf{PLTL} \models (A \,\mathcal{U}\,\mathsf{true}) \leftrightarrow \mathsf{true}$

- $\mathsf{PLTL} \models (A\,\mathcal{U}\,\mathsf{false}) \leftrightarrow \mathsf{false}$
- $\mathsf{PLTL} \models (\mathsf{true}\,\mathcal{U}\,A) \leftrightarrow \mathsf{F}\,A$
- $\mathsf{PLTL} \models (\mathsf{false}\,\mathcal{U}\,A) \leftrightarrow A$

**3.11 Remark** There are many similarities between the PLTL connectives $\mathsf{F}$, $\mathcal{U}$ and the modal connectives $\Diamond$ and $\Box$. In fact, some authors use the latter also for the temporal formulae. Note, however, that the linearity condition on the PLTL-models makes a difference. We have

- $\mathsf{PLTL} \models \mathsf{G}\,\mathsf{F}\,(A \vee C) \rightarrow (\mathsf{G}\,\mathsf{F}\,A \vee \mathsf{G}\,\mathsf{F}\,C)$ resp.
- $\mathsf{PLTL} \models (\mathsf{F}\,\mathsf{G}\,A \wedge \mathsf{F}\,\mathsf{G}\,C) \rightarrow \mathsf{F}\,\mathsf{G}\,(A \wedge C)$

while $\Box\Diamond(A \vee C) \rightarrow (\Box\Diamond A \vee \Box\Diamond C)$ and $(\Diamond\Box A \wedge \Diamond\Box p_q) \rightarrow \Diamond\Box(A \wedge C)$ are not valid in the modal logics K and S4.

**3.12 Definition** Formulae in negation normal form can be classified in the following way:

1. Propositional constants, literals and formulae of the form $\mathsf{X}\,A$ are called *elementary*.

2. All other formulae are called *non-elementary*. A non-elementary formula is said to be an $\alpha$-formula ($\beta$-formula) if it has a 'natural' representation as a conjunction (disjunction). The following tables list the possible forms of $\alpha$- and $\beta$-formulae and indicates the respective conjuncts $\alpha_1, \alpha_2$ and disjuncts $\beta_1, \beta_2$ (recall that $\overline{C}$ denotes the complement of $C$ which is equivalent to $\neg C$).

| $\alpha$ | $\alpha_1$ | $\alpha_2$ |
|---|---|---|
| $A \wedge C$ | $A$ | $C$ |
| $\mathsf{G}\,A$ | $A$ | $\mathsf{X}\,\mathsf{G}\,A$ |
| $A\,\mathcal{B}\,C$ | $\overline{C}$ | $A \vee \mathsf{X}\,(A\,\mathcal{B}\,C)$ |

| $\beta$ | $\beta_1$ | $\beta_2$ |
|---|---|---|
| $A \vee C$ | $A$ | $C$ |
| $\mathsf{F}\,D$ | $D$ | $\mathsf{X}\,\mathsf{F}\,D$ |
| $C\,\mathcal{U}\,D$ | $D$ | $C \wedge \mathsf{X}\,(C\,\mathcal{U}\,D)$ |

$\beta$-formulae of the form $\mathsf{F}\,D$ and $C\,\mathcal{U}\,D$ are also called *eventuality* formulae or *eventualities* for short; in order for these formulae to hold at a certain state in a model, there must be a future state where $D$ 'eventually' holds.

In the following we use $\alpha$, $\alpha_1$, $\alpha_2$ to denote an $\alpha$-formula and its conjuncts and $\beta$, $\beta_1$, $\beta_2$ to denote a $\beta$-formula and its disjuncts. Moreover, we assume for the rest of the chapter that there are no formulae of the form $\mathsf{F}\,D$, and (for symmetry) $\mathsf{G}\,A$. $\mathsf{F}\,D$ is equivalent to $\mathsf{true}\,\mathcal{U}\,D$, and $\mathsf{G}\,A$ to $\mathsf{false}\,\mathcal{B}\,\neg A$.

**3.13 Proposition** *We have for any $\alpha$-formula $\alpha$ and any $\beta$-formula $\beta$:*

*1.* $\mathsf{PLTL} \models \alpha \leftrightarrow (\alpha_1 \wedge \alpha_2)$.

*2.* $\mathsf{PLTL} \models \beta \leftrightarrow (\beta_1 \vee \beta_2)$.

*3.* $\alpha_1$, $\alpha_2$ *is in negation normal form iff* $\alpha$ *is in negation normal form.*

*4.* $\beta_1$, $\beta_2$ *is in negation normal form iff* $\beta$ *is in negation normal form.*

**3.14 Definition** We define the *closure* $cl(A)$ for any formula $A$ in negation normal form:

1. $A$ is in $cl(A)$.
2. If $\neg P$ is in $cl(A)$, then $P$ is in $cl(A)$.
3. If $\mathsf{X}\, B$ is in $cl(A)$, then $B$ is in $cl(A)$.
4. If $\alpha$ is in $cl(A)$, then $\alpha_1$ and $\alpha_2$ are in $cl(A)$.
5. If $\beta$ is in $cl(A)$, then $\beta_1$ and $\beta_2$ are in $cl(A)$.

The closure of a formula is essentially the set of all subformulae augmented with the $\alpha_2$ and $\beta_2$ parts of the temporal connectives. It is also called the Fischer-Ladner closure [28].

Before we turn to Hintikka structures for PLTL we define some properties for more general 'labeling' functions which assign to states sets of formulae rather than sets of variables.

**3.15 Definition** Let $S$ be a (possibly finite) sequence of states $s_0\, s_1\, \ldots$ and $L$ a function $L : S \to \mathrm{Pow(Fml)}$ and $s_i \in S$.

1. Propositional consistency properties:
   (PC0) $\mathsf{false}$ is not in $L(s_i)$.
   (PC1) If a literal $M$ is in $L(s_i)$, then its complement $\overline{M}$ is not in $L(s_i)$.
   (PC2) If $\alpha$ is in $L(s_i)$, then $\alpha_1$ *and* $\alpha_2$ are in $L(s_i)$.
   (PC3) If $\beta$ is in $L(s_i)$, then $\beta_1$ *or* $\beta_2$ is in $L(s_i)$.
2. Local consistency property:
   (LC) If $\mathsf{X}\, A$ is in $L(s_i)$ and $s_i$ is not the last state if $S$ is finite, then $A$ is in $L(s_{i+1})$.

We say that $L$ fulfills one of the above properties if the respective condition is satisfied for *all* states $s_i$ of the sequence.

In the next definition we describe the set of eventualities which are not 'satisfied' in a sequence of states.

**3.16 Definition** Let $S$ be a (possibly finite) sequence of states $s_0\, s_1\, \ldots$ and $L$ a function $L : S \to \mathrm{Pow(Fml)}$. Then the set $open(S, L)$ of eventualities is defined as:

$$open(S, L) := \{C\, \mathcal{U}\, D \mid \exists i\, (C\, \mathcal{U}\, D \in L(s_i)) \text{ and } \forall j \geq i\, (D \notin L(s_j))\}.$$

The following definition of a (pre-)Hintikka structure can be found in the literature (e.g. [22]).

**3.17 Definition** A *pre-Hintikka* structure $\mathcal{H}$ is a pair $\langle S, L\rangle$, where:

1. $S$ is a sequence of states $(s_i)_{i\in\mathbb{N}} = s_0\, s_1 \ldots$ .
2. $L : S \to \mathrm{Pow}(\mathrm{Fml})$ is a labeling function that fulfills the properties (PC0-3) and (LC).

By restricting the labeling function $L$ to variables, we can associate with each pre-Hintikka structure $\mathcal{H} = \langle S, L\rangle$ a model $\mathcal{M}_{\mathcal{H}} := \langle S, L\restriction\mathrm{Var}\rangle$.

**3.18 Definition** We say that a pre-Hintikka structure $\mathcal{H} = \langle S, L\rangle$ is a *Hintikka* structure if $open(S, L) = \emptyset$, that is, if we have for any state $s_i$ and any eventuality $C\,\mathcal{U}\,D$:

$$\text{If } C\,\mathcal{U}\,D \in L(s_i)\text{, then there exists a } j \geq i \text{ with } D \in L(s_j).$$

$\mathcal{H}$ is said to be a (pre-)Hintikka structure *for a formula* $A$ in negation normal form if $A \in L(s_0)$. We say that $\mathcal{H}$ is a *complete* (pre-)Hintikka structure for $A$ if for all $i$

$$L(s_i) = \{C \mid C \in cl(A) \text{ and } \mathcal{M}_{\mathcal{H}}, s_i \Vdash C\}.$$

Note that any Hintikka structure for $A$ can be made into a complete Hintikka structure for $A$ by adding to $L(s_i)$ all formulae of the closure satisfied at $s_i$. It is easy to see that the addition of these formulae preserves the properties of a Hintikka structure. The following standard theorem relates the existence of Hintikka structures to the existence of models.

**3.19 Theorem** *A formula $A$ in negation normal form is* PLTL*-satisfiable iff there exists a Hintikka structure for $A$.*

Proof: The direction from left to right is trivial since any model is a Hintikka structure as well. It only needs to be completed.

For the converse direction we assume that $\mathcal{H} = \langle S, L\rangle$ is a Hintikka structure for $A$. We show that the associated model $\mathcal{M}_{\mathcal{H}} = \langle S, L'\rangle$, where $L' = L\restriction\mathrm{Var}$ satisfies any formula $C \in L(s_i)$ at $s_i$, i.e. that $\mathcal{M}_{\mathcal{H}}, s_i \Vdash C$.

Assume that there exists a state $s_i$ and a formula $C \in L(s_i)$ with $\mathcal{M}_{\mathcal{H}}, s_i \nVdash C$. Assume further that $C$ has minimal literal length, i.e. that for any $j$ and for any formula $C'$ with $litlen(C') < litlen(C)$ and $C' \in L(s_j)$ we have $\mathcal{M}_{\mathcal{H}}, s_j \Vdash C'$.

$C$ cannot be a literal or a propositional constant since $L$ satisfies (PC0-1) and $L'(s_i) \subset L(s_i)$. $C$ cannot be of the form $\mathsf{X}\,C'$ since since $L$ satisfies (LC) and $litlen(C') < litlen(C)$. Therefore $C$ must be an $\alpha$- or a $\beta$-formula. $C$ cannot be of the form $C' \wedge C''$ or $C' \vee C''$ since $L$ satisfies (PC2-3) and $litlen(C')$ and $litlen(C'')$ are less than $litlen(C)$. We proceed by a case analysis:

$C = C' \mathcal{B} C''$ : Then $litlen(C') < litlen(C)$ and $litlen(\overline{C''}) < litlen(C)$ (see Proposition 2.5), and by induction on $j$ we can show that $C \in L(s_{i+j})$ and $\mathsf{X}\,(C' \mathcal{B} C'') \in L(s_{i+j})$ and therefore $\mathcal{M}_{\mathcal{H}}, s_{i+j} \Vdash \overline{C''}$ for all $j \geq 0$. Therefore $\mathcal{M}_{\mathcal{H}}, s_i \Vdash C' \mathcal{B} C''$. Contradiction.

$C = C' \mathcal{U} C''$ : Then $litlen(C') < litlen(C)$ and $litlen(C'') < litlen(C)$, and by induction on $j$ we can show that $C \in L(s_{i+j})$ and $\mathsf{X}\,(C' \mathcal{U} C'') \in L(s_{i+j})$ and $C'' \notin L(s_{i+j})$ for all $j \geq 0$. On the other hand, since $\mathcal{H}$ is a Hintikka structure, there must exist a $k \geq 0$ with $C'' \in L(s_{i+k})$. Contradiction.

□

**3.20 Convention** In the following we deal with a set $W$ of words over an alphabet $S$. We write $ws$ for the concatenation of a word $w$ and a single element $s \in S$. Similarly, we write $ww'$ for the concatenation of the two words $w$ and $w'$. $w$ and $w'$ may also be the empty word.

The following definition is new and cannot be found in the literature. We introduce a new type of structures which are essentially trees with loops on their branches. An application of the calculus which we present later will basically generate such trees.

**3.21 Definition** A *loop tree* is a tuple $\mathcal{T} = \langle W, S, L, R \rangle$ where:

1. $S$ is a finite set.
2. $W$ is a finite set of finite words on $S$ where:
   1. If $w = s_0 s_1 \ldots s_k \in W$, then $s_i \neq s_j$ for all $0 \leq i < j \leq k$.
2. $R$ is a binary relation on $W$ with the following properties:
   1. $(w, ws) \in R$ for all $w, ws \in W$.
   2. If $w \in W$ and $ws \notin W$ for all $s \in S$, then there exists a prefix $w' \in W$ of $w$ with $(w, w') \in R$.
   3. If $(w, w') \in R$ then either $w'$ is of the form $ws$ or $w'$ is a prefix of $w$.
4. $L : W \to \text{Pow}(\text{Fml})$ is a labeling function with the property:

$$L(ws) = L(w's) \text{ for all } ws, w's \in W.$$

The set $S$ can be viewed as a set of nodes and the words $W$ as directions how to reach these nodes. The conditions say that a word should contain a node only once, and that words which cannot be extended are related to a prefix. This means that we basically have a tree-like structure with loops back on the branches where at the end of each branch we have at least one loop back. The arrows in Fig. 6.3 correspond to the relation $R$. The labeling is controlled by the last node of a word. A word is essentially the last node plus the information how it is reached. Therefore words will also be called states.

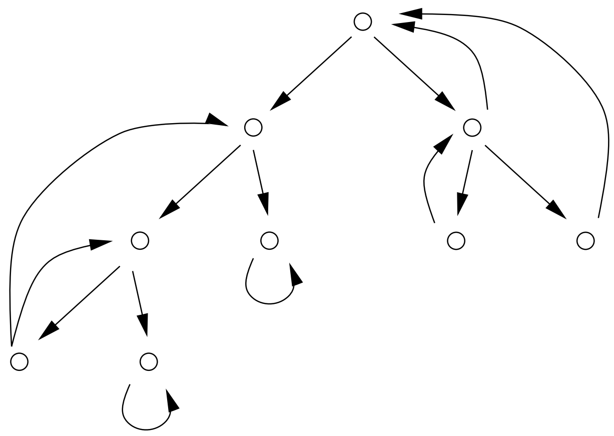


Figure 6.3: Example of a loop tree.

**3.22 Definition** Let $\mathcal{T} = \langle W, S, L, R \rangle$ be a loop tree.

1. If $ws \in W$ and $w \notin W$, then $ws$ is called a *root* of $\mathcal{T}$.

2. A *path* through $\mathcal{T}$ is a finite or infinite $R$-sequence of states $w_0, w_1, \ldots, w_i, w_{i+1}, \ldots$ where $(w_i, w_{i+1}) \in R$ for all $w_i$ of the sequence (except the last one if the sequence is finite).

3. A *loop branch* of $\mathcal{T}$ is a finite path $w_0, w_1, \ldots, w_k, w_{k+1}$ through $\mathcal{T}$ where $w_0$ is a root and for all $i < k$ $w_{i+1} = w_i s_i$ for some $s_i \in S$. The last state $w_{k+1}$ is identical to a previous state, i.e. $w_{k+1} = w_j$ for a $j \leq k$, and it is called the *loop state* of the branch. The suffix path $w_j, w_{j+1}, \ldots, w_k, w_{k+1}$ is called the *loop* of the branch. We say that a path $\pi$ *visits* the loop branch or simply the loop if $w_k, w_{k+1}$ occurs in $\pi$ (as a pair of consecutive states).

4. If $\pi = w_0, \ldots, w_j, w_{j+1}, \ldots, w_k, w_{k+1}$ is a loop branch, then the set $open_{\mathrm{inf}}(\pi, L)$ is defined as:

$$\begin{aligned} open_{\mathrm{inf}}(\pi, L) := \{C\, \mathcal{U}\, D \mid\ & C\, \mathcal{U}\, D \in open(\pi, L) \\ & \text{and } \forall i, (j \leq i \leq k \Rightarrow D \notin L(w_i))\}. \end{aligned}$$

**3.23 Remark**

- Note that a loop tree may contain several roots and may therefore represent several tree-like structures.

- A loop branch is defined to contain the backward loop. Therefore a 'physical branch' can contain several loop branches which share a common prefix path (see Fig. 6.3). Note that the definition of '$\pi$ visits the loop' refers only to the last state $w_k$ and the loop state $w_{k+1}$.

- Obviously, $open_{\inf}(\pi, L)$ is a subset of $open(\pi, L)$. It denotes the eventualities of $\pi$ which are not satisfied on the loop itself even if it is visited infinitely many times. In particular, if we expand $\pi$ into an infinite path $\pi'$, we have $open_{\inf}(\pi, L) = open(\pi, L)$.

**3.24 Definition** The length *pathlen* of a finite path $\pi$ is recursively defined:

1. If $\pi$ is of the form $w$, then $pathlen(\pi) := 1$.

2. If $\pi$ is of the form $\pi', w$, then $pathlen(\pi) := pathlen(\pi') + 1$.

**3.25 Definition** Let $W$ be the set of all words over an alphabet. Then the binary relation $\leq$ on $W$ is defined as follows:

$$w_1 \leq w_2 \text{ iff there exists a } w' \in W \text{ such that } w_2 = w_1 w'.$$

We also write $w_1 < w_2$ if $w_1 \leq w_2$ and $w_1 \neq w_2$. We further define the function $prefix : W \times W \to W$ as:

$$prefix(w_1, w_2) := \max\{\overline{w} \mid \overline{w} \leq w_1 \text{ and } \overline{w} \leq w_2\}.$$

Note that the prefix of two words of a loop tree always exists, but that it is not necessarily in the tree. However, the following lemma guarantees that if we have a path between two words, the prefix is in the tree and even on this path.

**3.26 Lemma** *Let $\mathcal{T} = \langle W, S, L, R\rangle$ be a loop tree. Then we have for any path $\pi = w_1, \ldots, w_n$ of length $n$ through $\mathcal{T}$ that $prefix(w_1, w_n)$ occurs in $\pi$.*

Proof: We prove the lemma by induction on the length $n$ of $\pi$.

$n = 1$ : Obvious.

$n = 2$ : That is, $\pi = w_1, w_2$ and $(w_1, w_2) \in R$. There are two possibilities: either $w_2 = w_1 s$ or $w_2 \leq w_1$. In the first case $prefix(w_1, w_2)$ is $w_1$ and in the second case $w_2$.

$n \to n + 1$ : Let $\pi = w_1, \ldots, w_n, w_{n+1}$ be a path of length $n + 1$. Then we know by the induction hypothesis that $\overline{w} := prefix(w_1, w_n)$ is on $w_1, \ldots, w_n$. We have again two cases:

a) $w_{n+1} = w_n s$: Then $\overline{w}$ is also a common prefix of $w_1$ and $w_{n+1}$ and if it is maximal, we are done. Otherwise, since $\overline{w}$ is maximal w.r.t. $w_1$ and $w_n$, $prefix(w_1, w_{n+1})$ must be $w_{n+1}$ which is trivially on $\pi$.

b) $w_{n+1} \leq w_n$ (loop): Then either $\overline{w} \leq w_{n+1}$ or $w_{n+1} < \overline{w}$. In the first case $prefix(w_1, w_{n+1})$ is also $\overline{w}$ and in the second case $w_{n+1}$. Both states are on $\pi$.

□

**3.27 Lemma** *Let $\pi = w, \ldots, w''$ be a finite path with $w \leq w''$. Then all states $w'$ with $w \leq w' \leq w''$ occur in $\pi$.*

Proof: We prove the lemma by induction on the number $n$ of loops that $\pi$ visits, i.e. on the number of consecutive pairs $w_k, w_{k+1}$ with $w_k \geq w_{k+1}$.

$n = 0$: If no loop is visited, then any consecutive pair on $\pi$ must be of the form $w', w's$ and the claim is trivially satisfied.

$n \to n+1$: Starting from $w$, we follow $\pi$ as long as the states are prefixes of $w''$ and no loop occurs. Let $w'$ be the last such state, i.e. $w' \leq w''$ and $\pi$ begins with all states between $w$ and $w'$. If the next state loops back, that is, if it is a prefix of $w'$, we can apply the induction hypothesis and the claim follows. Otherwise, the next state has the form $w's$ with $w's \not\leq w''$. Obviously, $\mathit{prefix}(w's, w'') = w'$ and by the previous lemma we know that $w'$ occurs again in the path $w's, \ldots, w''$. In the section from $w's$ to the second occurrence of $w'$, we must visit at least one loop since $w' < w's$. Therefore, we can apply the induction hypothesis to the remaining path from the second occurrence of $w'$ to $w''$ where at most $n$ loops may be visited. $\square$

**3.28 Corollary** *If $w_j, w_{j+1}, \ldots, w_k, w_{k+1}$ is a loop and a path $\pi$ visits it repeatedly (i.e. more than one occurrence of $w_k, w_{k+1}$ in $\pi$), then obviously all other states of the loop $w_{j+1}, \ldots, w_{k-1}$ must occur between two occurrences of $w_k, w_{k+1}$, although not necessarily in a row.*

Proof: The corollary follows from the lemma since we have a path from $w_j$ to $w_k$ and $w_j$ is a prefix of $w_k$. $\square$

**3.29 Definition** Let $\mathcal{T} = \langle W, S, L, R \rangle$ be a loop tree. The *subtree of $\mathcal{T}$ at $w \in W$* is a structure $\mathcal{T}' = \langle W', S', L', R' \rangle$ defined as follows:

1. $S' := S$.
2. $W' := \{ww' \mid ww' \in W\}$.
3. $R' := \{(ww', ww'') \mid (ww', ww'') \in R\}$.
4. $L' := L{\restriction}W'$.

We say that $\mathcal{T}'$ is an *isolated* subtree of $\mathcal{T}$ if $(w', v) \notin R$ for any $w' \in W'$ and $v \in W \setminus W'$.

**3.30 Proposition** *An isolated subtree of a loop tree is also a loop tree.*

We are interested in isolated subtrees because obviously a path entering an isolated subtree cannot leave it anymore. Later, when we look for a satisfying path in a loop tree, we have the alternative that isolated subtrees either contain a solution, or we can discard them.

**3.31 Definition** Let $\mathcal{T} = \langle W, S, L, R \rangle$ be a loop tree. The function $\text{depth}_{\mathcal{T}} : W \to \mathbb{N}$ is defined as follows:

1. $\text{depth}_{\mathcal{T}}(w) := 0$ for any root $w$ of $\mathcal{T}$.

2. $\text{depth}_{\mathcal{T}}(ws) := \text{depth}_{\mathcal{T}}(w) + 1$ for any $w$, $ws \in W$.

Whether or not a subtree is isolated can be determined by verifying the loop states of the loop branches passing through the subtree's root.

**3.32 Lemma** *Let $\mathcal{T} = \langle W, S, L, R \rangle$ be a loop tree and $\mathcal{T}' = \langle W', S', L', R' \rangle$ the subtree at $w \in W$. Then we have: $\mathcal{T}'$ is isolated iff*

$$\text{depth}_{\mathcal{T}}(w) \leq \min(\{\text{depth}_{\mathcal{T}}(w') \mid \begin{array}{l} w' \textit{is a loop state of a} \\ \textit{loop branch of } \mathcal{T} \textit{ containing } w\}). \end{array}$$

Proof: If $\mathcal{T}'$ is isolated, then no loop branch of $\mathcal{T}$ containing $w$ can have a loop state outside $\mathcal{T}'$. Since $w$ is the root of $\mathcal{T}'$, the depth of a loop state must be greater or equal than the depth of $w$.

Conversely, if the depth of a loop state is greater or equal than the depth of $w$, then it must belong to $\mathcal{T}'$ since a branch may only loop back on itself. Therefore $\mathcal{T}'$ must be isolated. □

**3.33 Definition**

1. A *pre-Hintikka-tree* is a loop tree $\mathcal{T} = \langle W, S, L, R \rangle$ where $L$ fulfills the properties (PC0-3) and (LC) for all infinite paths $\pi = w_0, w_1, \ldots$ through $\mathcal{T}$.

2. A *Hintikka-tree* for a formula $A$ is a pre-Hintikka-tree $\mathcal{T} = \langle W, S, L, R \rangle$ with the additional property that there exists an infinite path $\pi = w_0, w_1, \ldots$ through $\mathcal{T}$ with $A \in L(w_0)$ and $open(\pi, L) = \emptyset$.

**3.34 Lemma** *Let $\pi = w_j, w_{j+1}, \ldots, w_k, w_{k+1}$ be a loop ($w_{k+1} = w_j$) of a pre-Hintikka-tree $\mathcal{T} = \langle W, S, L, R \rangle$. Then we have: If an eventuality $C \,\mathcal{U}\, D$ is in $open_{\text{inf}}(\pi, L)$, then $C \,\mathcal{U}\, D$ and $\mathsf{X}\,(C \,\mathcal{U}\, D)$ are in $L(w_i)$ for all $i$ with $j \leq i \leq k$.*

Proof: First we show that if $C \,\mathcal{U}\, D \in L(w_i)$ ($j \leq i \leq k$), then $C \,\mathcal{U}\, D$ and $\mathsf{X}\,(C \,\mathcal{U}\, D)$ are in $L(w_{i'})$ for all $i'$ with $i \leq i' \leq k$. This is done by straightforward induction on $i' - i$ using that $C \,\mathcal{U}\, D$ is a $\beta$-formula, that $D$ is not in $L(w_{i'})$ and that $L$ fulfills (PC3) and (LC).

Therefore $C \,\mathcal{U}\, D$ and $\mathsf{X}\,(C \,\mathcal{U}\, D)$ are in $L(w_k)$ and by the same arguments we have that $C \,\mathcal{U}\, D$ and $\mathsf{X}\,(C \,\mathcal{U}\, D)$ are in $L(w_{k+1})$ since $(w_k, w_{k+1}) \in R$. Applying the just shown proposition to $w_j = w_{k+1}$ concludes the proof. □

**3.35 Corollary** *Let $\pi = w_j, w_{j+1}, \ldots, w_k, w_{k+1}$ be a loop of a pre-Hintikka-tree $\mathcal{T} = \langle W, S, L, R \rangle$. Then we have: If an eventuality $C \,\mathcal{U}\, D$ is in $open_{\text{inf}}(\pi, L)$, then $\mathsf{X}\,(C \,\mathcal{U}\, D)$ is in the label of the loop state $w_{k+1}$.*

**3.36 Lemma** *Let $\pi$ be an infinite path through the pre-Hintikka-tree $\mathcal{T} = \langle W, S, L, R\rangle$. Then we have:*

*a) $\pi$ visits only finitely many loops $\pi_1, \ldots, \pi_m$ infinitely many times.*

*b) There exists a suffix path $\pi'$ of $\pi$ which contains only states from $\pi_1, \ldots, \pi_m$.*

*c) All states of $\pi_1, \ldots, \pi_m$ occur infinitely many times in $\pi$.*

Proof:

a) Trivial since $W$ is finite.

b) Assume that there are infinitely many states in $\pi$ which are not in one of $\pi_1, \ldots, \pi_m$. Since $W$ is finite, there must exist at least one state $w$ which occurs infinitely many times but which is not in $\pi_1, \ldots, \pi_m$. After each occurrence of $w$ the path $\pi$ must visit a loop with loop state $w' \leq w$. Again, because of the finiteness of the tree, $\pi$ must visit at least one such loop infinitely many times. That is, $w$ is in a loop $w', \ldots, w, \ldots, w'$ which is visited infinitely many times and which must therefore be one of $\pi_1, \ldots, \pi_m$. Contradiction.

c) Follows directly from corollary 3.28.

□

The following lemma states that the open eventualities of a path depend in a simple way on the unfulfilled eventualities of single loop branches.

**3.37 Lemma** *Let $\pi$ be an infinite path through the pre-Hintikka-tree $\mathcal{T} = \langle W, S, L, R\rangle$ and $\pi_1, \ldots, \pi_m$ be the loops of $\mathcal{T}$ that are visited infinitely many times by $\pi$. Then we have:*

$$\mathit{open}(\pi, L) = \bigcap_{i=1\ldots m} \mathit{open}_{\inf}(\pi_i, L).$$

Proof:

$\supset$: Let $C \,\mathcal{U}\, D$ be in $\bigcap_{i=1\ldots m} \mathit{open}_{\inf}(\pi_i, L)$. From Lemma 3.36.b) we know that there is a point after which only states from $\pi_1, \ldots, \pi_m$ are visited. If $C \,\mathcal{U}\, D \in \mathit{open}_{\inf}(\pi_i, L)$, then $D$ is not in any state of $\pi_i$, and by Lemma 3.34 we know that $C \,\mathcal{U}\, D$ is in each state of $\pi_i$. Therefore $C \,\mathcal{U}\, D$ must be in $\mathit{open}(\pi, L)$.

$\subset$: Let $C \,\mathcal{U}\, D$ be in $\mathit{open}(\pi, L)$. Then there is a state $s$ in $\pi$ such that for any future state $s'$ the formula $D$ is not in $L(s')$ but $\mathsf{X}\,(C \,\mathcal{U}\, D)$ is in $L(s')$. This implies that for any state $s''$ from $\pi_1, \ldots, \pi_m$ the formula $D$ is not in $L(s'')$ and $\mathsf{X}\,(C \,\mathcal{U}\, D)$ is in $L(s'')$ since by Lemma 3.36.c) all these states are visited by $\pi$ after $s$. Therefore $C \,\mathcal{U}\, D$ is in $\mathit{open}_{\inf}(\pi_i, L)$ for all $\pi_i$ $(i = 1 \ldots m)$. □

**3.38 Theorem** *There is a Hintikka structure for a formula $A$ iff there exists a Hintikka-tree for $A$.*

Proof: The direction from right to left is obvious. If $\mathcal{T} = \langle W, S, L, R\rangle$ is a Hintikka-tree for $A$ then simply choose a path $\pi = w_0, w_1, \ldots$ through $\mathcal{T}$ with $A \in L(w_0)$ and $open(\pi, L) = \emptyset$. $\langle \pi, L\rangle$ is then a Hintikka structure for $A$.

For the direction from left to right assume that $\mathcal{H} = \langle S, L\rangle$ is a Hintikka structure for $A$ with

$$S = s_0 s_1 \ldots s_i s_{i+1} \ldots \ .$$

First, we introduce an equivalence relation $\sim$ on the elements of $S$:

$$s_i \sim s_j \text{ iff } L(s_i) \cap cl(A) = L(s_j) \cap cl(A).$$

The equivalence class of $s_i$ is denoted by $[s_i]$. We construct a Hintikka-tree $\mathcal{T} = \langle W, S', L', R\rangle$ for $A$ in the following way ($w, w', w''$ may be the empty word):

1. $S' := S/{\sim}$.
2. $W$ and $R$ are defined inductively:
   1. $[s_0]$ is an element of $W$.
   2. If $w[s_i]$ is an element of $W$, then we distinguish two cases:
      a) If $w[s_i]$ contains a state equivalent to $s_{i+1}$, that is, if $w[s_i] = w'[s_j]w''$ and $s_j \sim s_{i+1}$, then $(w[s_i], w'[s_j])$ is in $R$ (a loop).
      b) Otherwise $w[s_i][s_{i+1}]$ is an element of $W$ and $(w[s_i], w[s_i][s_{i+1}])$ is in $R$.
3. The labeling $L'$ is defined as $L'(w[s_i]) := L(s_i) \cap cl(A)$.

The structure $\mathcal{T}$ is obviously a loop tree. $S'$ is finite since $cl(A)$ is finite, $L'$ satisfies (PC0-3) and (LC), and by the construction there is a path $\pi = w_0, w_1, \ldots$ through $\mathcal{T}$ (corresponding to $s_0 s_1 \ldots$) with $w_0 = [s_0]$, $A \in L'(w_0)$ and $open(\pi, L') = \emptyset$. Therefore $\mathcal{T}$ is a Hintikka-tree for $A$. $\square$

## 6.4 The Calculus $\mathsf{PLTL}_T$

We present a Tableau-like calculus for $\mathsf{PLTL}$ which is complete and correct with respect to the $\mathsf{PLTL}$ semantics. It operates on so-called prestates which contain the full information needed to decide satisfiability of formulae in negation normal form.

**4.1 Convention** In the following we use $\Gamma$ and $\Sigma$ for finite sets of formulae in negation normal form, and $\Lambda$ for sets of *literals* (and possibly constants). We also write $A, \Gamma$ for the set $\{A\} \cup \Gamma$, and $\Gamma, \Sigma$ for the union $\Gamma \cup \Sigma$, and $\mathsf{X}\,\Gamma$ is used for the set $\{\mathsf{X}\,A \mid A \in \Gamma\}$.

For lists we have the following conventions: We use $*$ for the concatenation of lists and [] for the empty list. If $M$ is a list, then we write $len(M)$ for the length of $M$ and $M[i]$ for the $i^{\text{th}}$ element of $M$ $(1 \leq i \leq len(M))$. If $M$ is a list of tuples, then we write $M[i]_j$ to denote the projection to the $j^{\text{th}}$ element of $M[i]$.

**4.2 Definition** A *prestate* is a triple $(\Gamma, \mathit{Save}, \mathit{Res})$, also written as $\Gamma \mid \mathit{Save} \mid \mathit{Res}$ where:

1. $\Gamma$ is a finite set of formulae in negation normal form.
2. *Save* is a structure to store history information. It is a pair $(\mathit{Ev}, \mathit{Br})$, also written as $\mathit{Ev}\,;\,\mathit{Br}$ where:
    1. *Ev* is a set of formulae in negation normal form. It represents the currently satisfied eventualities.
    2. *Br* is a list of pairs $(\Gamma', \mathit{Ev}')$ representing the current branch. Both $\Gamma'$ and $\mathit{Ev}'$ are sets of formulae in negation normal form and correspond to the $\Gamma$ and *Ev* parts of previous prestates.
3. *Res* is a structure to store partial result information. It is a pair $(n, \mathit{uev})$ where:
    1. $n$ is a natural number indicating the 'earliest' prestate reachable by the current one.
    2. *uev* is a set of eventuality formulae in negation normal form. It represents the unfulfilled eventualities of the current branch.

A prestate is said to be a *state* if $\Gamma$ is of the form $\Lambda, \mathsf{X}\,\Sigma$, that is, if $\Gamma$ consists only of elementary formulae.

**4.3 Convention** According to the above definition, $\Gamma \mid \mathit{Ev}\,;\,\mathit{Br} \mid (n, \mathit{uev})$ is the extended notion for an abstract prestate. To focus on the locally relevant parts of a prestate, we use '...' for the 'unimportant' parts (e.g. $\Gamma \mid \ldots \mid \ldots$). If '...' appears at the same position in the numerator and the denominator(s) of a rule, then we mean that the corresponding parts are the same.

**4.4 Definition** The Tableau calculus $\mathsf{PLTL}_T$ is defined as follows:

a) Terminal rules:

$$\mathsf{false}, \Gamma \mid Ev \,;\, Br \mid (len(Br), \{\mathsf{false}\}) \qquad (\mathsf{false})$$
$$P, \neg P, \Gamma \mid Ev \,;\, Br \mid (len(Br), \{\mathsf{false}\}) \qquad (\text{contr})$$
$$\Lambda, \mathsf{X}\,\Sigma \mid Ev \,;\, Br \mid (k, uev) \qquad (\text{loop})$$

where in (loop) there exists an $i$, $1 \leq i \leq len(Br)$, such that:

1. $\Lambda, \mathsf{X}\,\Sigma = Br[i]_1$.
2. $k = i - 1$ and $uev = \{C\,\mathcal{U}\,D \mid C\,\mathcal{U}\,D \in \Sigma$ and $D \notin (\cup_{j=i+1}^{len(Br)} Br[j]_2 \cup Ev)\}$.

b) $\alpha$-rules:

$$\frac{\alpha, \Gamma \mid \ldots \mid \ldots}{\alpha_1, \alpha_2, \Gamma \mid \ldots \mid \ldots}\ (\alpha)$$

c) $\beta$-rules:

$$\frac{A \vee C, \Gamma \mid \ldots ;\, Br \mid (n, uev)}{A, \Gamma \mid \ldots ;\, Br \mid (n_1, uev_1) \qquad C, \Gamma \mid \ldots ;\, Br \mid (n_2, uev_2)}\ (\vee)$$

$$\frac{C\,\mathcal{U}\,D, \Gamma \mid Ev \,;\, Br \mid (n, uev)}{D, \Gamma \mid Ev \cup \{D\} \,;\, Br \mid (n_1, uev_1) \qquad C, \mathsf{X}\,(C\,\mathcal{U}\,D), \Gamma \mid Ev \,;\, Br \mid (n_2, uev_2)}\ (\mathcal{U})$$

where in $(\vee)$ and $(\mathcal{U})$ :

1. $n = \min(n_1, n_2)$.
2. $(m := len(Br) - 1)$.

$$uev = \begin{cases} \emptyset & \text{if } uev_1 = \emptyset \text{ or } uev_2 = \emptyset, \\ \{\mathsf{false}\} & \text{if } n_1 > m \text{ and } n_2 > m \text{ (and } uev_1 \neq \emptyset,\ uev_2 \neq \emptyset), \\ uev_1 & \text{if } n_1 \leq m \text{ and } n_2 > m \text{ (and } uev_2 \neq \emptyset), \\ & \text{or if } uev_2 = \{\mathsf{false}\}, \\ uev_2 & \text{if } n_1 > m \text{ and } n_2 \leq m \text{ (and } uev_1 \neq \emptyset), \\ & \text{or if } uev_1 = \{\mathsf{false}\}, \\ uev_1 \cap uev_2 & \text{otherwise.} \end{cases}$$

d) Nexttime rule:

$$\frac{\Lambda, \mathsf{X}\,\Sigma \mid Ev \,;\, Br \mid \ldots}{\Sigma \mid \emptyset \,;\, Br * ((\Lambda, \mathsf{X}\,\Sigma), Ev) \mid \ldots}\ (\mathsf{X})$$

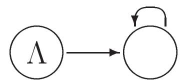
In order to ensure termination, the $\alpha$- and $\beta$-rules and the nexttime rule are restricted to prestates which are not instances of a terminal rule.

We call $\alpha$ in ($\alpha$), $A \vee B$ in ($\vee$), and $C\,\mathcal{U}\,D$ in ( $\mathcal{U}$ ) the *decomposed formula* of the respective rule.

**4.5 Remark**

- The main difference to a modal calculus is the result part which is synthesized bottom-up (from children to parents, see Fig. 6.7). It is needed because a single branch need not be 'open' or 'closed'; it may be 'open' in connection with some other branches.
- (loop): $Br[i]_1, Br[i+1]_1, \ldots Br[len(Br)]_1, \Lambda, \mathsf{X}\,\Sigma$ correspond to the loop of a branch. $uev$ is defined to be the set $open_{\text{inf}}(.,.)$ of eventualities which are not satisfied on this loop branch (see corollary 3.35). $i$ is set to the depth of the state to which the branch loops back.
- ($\beta$-rules): A $\beta$-rule corresponds to a branching of the tree. $n$ is set to the minimum depth of the states to which branches of the two subtrees can loop back. $uev$ is the minimal set of eventualities left open by any infinite path visiting only loops of the subtree below this $\beta$ node. The 'main' case in which both branches are used is the last clause. See also the remark after corollary 4.12.
- (Nexttime rule): The current state and the eventualities satisfied by the current state are appended to the branch $Br$.
- Note that the sets $\Gamma$, $\Lambda$, and $\Sigma$ may be empty. For instance, if $\Sigma$ is empty in the numerator of ( $\mathsf{X}$ ), we obtain the following fragment of a tableau which ends in a basically empty instance of (loop). On the right the corresponding model is shown.

$$\dfrac{\dfrac{\Lambda \mid Ev\,;\ Br \mid \ldots}{\emptyset \mid \emptyset\,;\ Br * (\Lambda, Ev) \mid \ldots}\ (\mathsf{X})}{\emptyset \mid \emptyset\,;\ Br * (\Lambda, Ev) * (\emptyset, \emptyset) \mid (len(Br)+1, \emptyset)\ \ \text{(loop)}}\ (\mathsf{X})$$




**4.6 Definition** A *tableau* for a prestate $ps$ is a tree of prestates with root $ps$ and in which the sons of a node (prestate) correspond to an application of a $\mathsf{PLTL}_T$ rule to the node. We say that the tableau is *expanded*, if each leaf node is an instance of a terminal rule.

Let $A$ be a formula and $n$ the number of subformulae of $A$. Then it is clear that any tableau for $A \mid \ldots \mid \ldots$ is finite. There are $2^{O(n)}$ many subsets of $cl(A) \cup cl(\overline{A})$. Each $\Gamma$ of a prestate $\Gamma \mid \ldots \mid \ldots$ is such a subset, and since the terminal rules must be applied whenever they can be applied, the number of different prestates on each branch is finite. Therefore the total number of prestates in any tableau for $A \mid \ldots \mid \ldots$ is finite and any expansion will eventually terminate.

**4.7 Lemma**

*a) For every formula $A$ there is an $n \in \mathbb{N}$ and a set uev $\subseteq$* Fml *such that there is an expanded tableau for $A \,|\, \emptyset\,;\, Br \,|\, (n, uev)$.*

*b) If in a tableau the set uev of a prestate is empty , then the set uev of the root of the tableau is also empty.*

Proof:

a) Let $\mathsf{PLTL}_T{}'$ be the calculus $\mathsf{PLTL}_T$ restricted to prestates without *Res* part. Construct any expanded $\mathsf{PLTL}_T{}'$ tableau for $A \,|\, \emptyset\,;\, Br$. Convert the $\mathsf{PLTL}_T{}'$ tableau into an expanded $\mathsf{PLTL}_T$ tableau by adding from the leaves to the root the *Res* part according to the rules of $\mathsf{PLTL}_T$.

b) The condition in the $\beta$-rules ensures that the empty set is propagated towards the root.

$\square$

**4.8 Example** We show the essential branch of a tableau for $\mathsf{G}\,\mathsf{F}\,p \wedge \mathsf{G}\,\mathsf{F}\,\neg p$, that is, for the satisfiable property that there is always a state in the future in which $p$ holds, and that there is always a state in the future in which $\neg p$ holds. Recall that $\mathsf{F}\,p$ can be written as $\mathsf{true}\,\mathcal{U}\,p$ (as for $\mathsf{G}$, it is simpler to assume that it is not expressed in terms of $\mathcal{B}$).

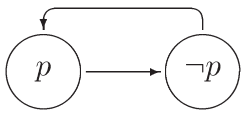
The $\alpha$- and $\beta$-rules are applied until we reach a state with only elementary formulae. The currently decomposed formula is in parentheses. It is left to the reader to fill in the missing *Save* and *Res* parts.

$$(\mathsf{G}\,\mathsf{F}\,p \wedge \mathsf{G}\,\mathsf{F}\,\neg p) \,|\, \ldots \,|\, \ldots$$
$$(\mathsf{G}\,\mathsf{F}\,p), \mathsf{G}\,\mathsf{F}\,\neg p \,|\, \ldots \,|\, \ldots$$
$$\mathsf{F}\,p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,p, (\mathsf{G}\,\mathsf{F}\,\neg p) \,|\, \ldots \,|\, \ldots$$
$$(\mathsf{F}\,p), \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,p, \mathsf{F}\,\neg p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,\neg p \,|\, \ldots \,|\, \ldots$$
$$p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,p, (\mathsf{F}\,\neg p), \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,\neg p \,|\, \{p\}\,;\,.\,|\, \ldots \qquad \mathrm{Sub}_1$$
$$p, \neg p, \ldots \qquad \boxed{p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,p, \mathsf{X}\,\mathsf{F}\,\neg p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,\neg p \,|\, \{p\}\,;\,.\,|\, \ldots} \quad (\mathsf{X})$$
$$(\mathsf{G}\,\mathsf{F}\,p), \mathsf{F}\,\neg p, \mathsf{G}\,\mathsf{F}\,\neg p \,|\, \ldots \,|\, \ldots$$
$$\mathsf{F}\,p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,p, \mathsf{F}\,\neg p, (\mathsf{G}\,\mathsf{F}\,\neg p) \,|\, \ldots \,|\, \ldots$$
$$(\mathsf{F}\,p), \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,p, \mathsf{F}\,\neg p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,\neg p \,|\, \ldots \,|\, \ldots$$
$$\mathrm{Sub}_2 \qquad \mathsf{X}\,\mathsf{F}\,p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,p, (\mathsf{F}\,\neg p), \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,\neg p \,|\, \ldots \,|\, \ldots$$
$$\boxed{\mathsf{X}\,\mathsf{F}\,p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,p, \neg p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,\neg p \,|\, \{\neg p\}\,;\,.\,|\, \ldots} \qquad \mathrm{Sub}_3 \quad (\mathsf{X})$$
$$\mathsf{F}\,p, (\mathsf{G}\,\mathsf{F}\,p), \mathsf{G}\,\mathsf{F}\,\neg p \,|\, \ldots \,|\, \ldots$$
$$\mathsf{F}\,p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,p, (\mathsf{G}\,\mathsf{F}\,\neg p) \,|\, \ldots \,|\, \ldots$$
$$(\mathsf{F}\,p), \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,p, \mathsf{F}\,\neg p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,\neg p \,|\, \ldots \,|\, \ldots$$
$$p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,p, (\mathsf{F}\,\neg p), \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,\neg p \,|\, \{p\}\,;\,.\,|\, \ldots \qquad \mathrm{Sub}_4$$
$$p, \neg p, \ldots \qquad p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,p, \mathsf{X}\,\mathsf{F}\,\neg p, \mathsf{X}\,\mathsf{G}\,\mathsf{F}\,\neg p \,|\, \ldots \,|\, (0, \emptyset) \quad \text{(loop)}$$

The highlighted prestates above the $(\mathsf{X})$'s are the *states* of the tableau; the first one (at 'state depth' 0) satisfies $p$ and the second one satisfies $\neg p$. The essential branch ends in an instance of (loop), where in the *Res* part the 0 refers to the depth 0 of the first state and the $\emptyset$ indicates that all eventualities (the only candidate to check stems from $\mathsf{X}\,\mathsf{F}\,\neg p$) are satisfied on this loop.

The subtableau $\mathrm{Sub}_1$ is essentially a copy of the presented tree with $p$ and $\neg p$ exchanged. The subtableau $\mathrm{Sub}_2$ detects a loop to the first state; however, on this shortcut the second eventuality $\mathsf{F}\,\neg p$ is left open. We have the same situation in subtableau $\mathrm{Sub}_4$ with the second state and the first eventuality. The subtableau $\mathrm{Sub}_3$ is simply a 'detour' since the fulfillment of both eventualities is delayed to the future.

The corresponding model is very simple:




**4.9 Example** In the following example, we have two essential branches. We try to satisfy the formula

$$\begin{array}{l}(\mathsf{G}\,\mathsf{F}\,p_0 \wedge \mathsf{G}\,\mathsf{F}\,p_1) \wedge \\ \mathsf{G}\,(p_0 \rightarrow (\neg p_1 \wedge \mathsf{X}\,\neg p_1)) \wedge \\ \mathsf{G}\,(p_1 \rightarrow (\neg p_0 \wedge \mathsf{X}\,\neg p_0)) \wedge \\ (\neg p_0 \wedge \neg p_1).\end{array}$$

We only show the skeleton of the tableau, that is, the states and the instances of (loop) of the two essential branches. To simplify the figure further, we omit the nexttime formulae (the set $\mathsf{X}\,\Sigma$ in $(\mathsf{X})$ and (loop)).

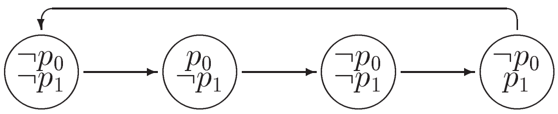
$$\neg p_0, \neg p_1 \mid \ldots \mid (0, \emptyset)$$

$$p_0, \neg p_1 \mid \ldots \mid (0, \{\mathsf{F}\,p_1\}) \qquad\qquad \neg p_0, p_1 \mid \ldots \mid (0, \{\mathsf{F}\,p_0\})$$

$$\neg p_0, \neg p_1 \mid \ldots \mid (0, \{\mathsf{F}\,p_1\}) \ \text{(loop)} \qquad\qquad \neg p_0, \neg p_1 \mid \ldots \mid (0, \{\mathsf{F}\,p_0\}) \ \text{(loop)}$$

Figure 6.4: Skeleton of the tableau of example 4.9.

The corresponding model is a linearization of the two branches. For better readability not only the variables but also negative literals which are true at a certain state are displayed (i.e. the respective variable is false at that state).




**4.10 Definition** The *loop tree* $\mathcal{T} = \langle W, S, L, R\rangle$ *for an expanded tableau* is defined in the following way:

1. $S$ is the set of all states (not prestates!) of the tableau and the set of leaf nodes which are not instances of (loop).

2. $W$ is the set of paths (in terms of $S$) to the elements of $S$ in the tableau.

3. $R$:
   1. $(w, ws)$ is in $R$ for all $w, ws \in W$.
   2. $(ws, ws)$ is in $R$ if $s$ is an instance of (false) or (contr). That is, we draw a loop to the last state itself if it is inconsistent ((PC0) or (PC1) is violated).
   3. If $w \in W$ is a path to a state which is the last state before an instance of (loop) in the tableau, then $w$ must be of the form $w'sw''$ where $s$ is the referenced loop state. Then $(w, w's)$ is in $R$.

4. $L(ws)$ is the set $\Gamma$ if $s = \Gamma \mid \ldots \mid \ldots$ plus all the formulae that are decomposed in the tableau between $w$ and $ws$.

We could also (formally) omit the classical contradictions from the loop tree (and the states which have only contradictory prestates below), and we would obtain a pre-Hintikka-tree. However, the relevant information is always in the result part.

**4.11 Lemma** *Let $\mathcal{T} = \langle W, S, L, R\rangle$ be the loop tree for an expanded tableau. Then we have:*

*a) $L$ fulfills (PC0-3) and (LC) for any state $ws \in W$ where $s$ is not an instance of* (false) *or (contr).*

*b) For all $ws \in W$, $s = \Gamma \mid Ev\,;\, Br \mid \ldots$, we have* $\mathrm{depth}_{\mathcal{T}}(ws) = len(Br)$.

*c) For all $ws \in W$, $s = \Lambda, \mathsf{X}\,\Sigma \mid Ev\,;\, Br \mid (n, uev)$, we have*

$$n = \min(\{\mathrm{depth}_{\mathcal{T}}(v) \mid v \textit{ is a loop state of a branch } \pi = \ldots ws \ldots\}.$$

Proof:

a) Let $s' = \Gamma \mid \ldots \mid \ldots$ be a prestate between $w$ and $ws$. Since $s$ is not an instance of (false) or (contr) and $s$ is a state, any formula $\alpha\,(\beta) \in \Gamma$ will be decomposed (i.e. $\Gamma \subseteq L(ws)$), as otherwise, $\mathcal{T}$ would not be an expanded tableau.

b) Induction on $i = \mathrm{depth}_{\mathcal{T}}(ws)$. $i = 0$: that is, $ws$ is a root, and $s$ must be of the form $\Gamma \,|\, Ev\,;\, [] \,|\, \ldots$ . Inductive step, $i \to i+1$: then $\mathrm{depth}_{\mathcal{T}}(ws) = i + 1$, $\mathrm{depth}_{\mathcal{T}}(w) = i$ and $w$ is of the form $w's'$, $s' = \Gamma \,|\, Ev\,;\, Br \,|\, \ldots$ where by the induction hypothesis $i = len(Br)$. The state $s$ is a successor of $s'$ and has therefore the form $\ldots \,|\, Ev'\,;\, Br * (\Gamma, Ev) \,|\, \ldots$, i.e. $len(Br * (\Gamma, Ev)) = i + 1$.

c) Let $j$ be the length of the longest branch $\pi = \ldots ws \ldots$, and let $i = \mathrm{depth}_{\mathcal{T}}(ws)$. We proceed by induction on $k = j - i$.

   $k = 0$: Then $i = j$ and $s$ is either an instance of (false) or (contr), or it is the last state before an instance of (loop). In the first case, $s$ is a loop state and the claim follows by the definition of (false) and (contr), and b). In the second case, note that $s$ may be the 'father' of several instances of (loop). Each such instance determines the depth of a loop state and the conditions on the $\beta$-rules ensure that the $n$ in $s$ is the minimum of these depths.

   $k \to k + 1$: Then in a branch $\pi = \ldots ws \ldots$ the state $ws$ may again be the last state before instances of terminal rules, and for all successors $wss'$ the claim is true by the induction hypothesis. For either case the conditions on the $\beta$-rules ensure that in $s$ we have the minimum of depths.

□

**4.12 Corollary** *Let $\mathcal{T} = \langle W, S, L, R\rangle$ be the loop tree for an expanded tableau. Then we have for all $ws \in W$, $s = \Gamma \,|\, Ev\,;\, Br \,|\, (n, uev)$*

*The subtree of $\mathcal{T}$ at $ws$ is isolated iff $n \geq len(Br)$.*

Proof: Follows from b) and c) of the lemma above and Lemma 3.32. □

Note that the $\beta$-rule applications in the tableau are between two states. The lower state is at depth $len(Br)$. The conditions in the $\beta$-rules, however, control the result synthesis for the upper state which is at depth $len(Br) - 1$. If a denominator of a $\beta$-rule application leads to an isolated subtree with a non-empty set $uev$, this subtree can be discarded (clauses 2-4 in the condition for $uev$ in the $\beta$-rule). If both subtrees are discarded (clause 2), $uev$ is set to $\{\mathsf{false}\}$ to indicate that any path through the numerator will have unfulfilled eventualities.

**4.13 Theorem** *(Correctness)*
*If $A$ is a formula in negation normal form and if there exists an $n$ such that there is an expanded tableau for $A \,|\, \emptyset\,;\, [] \,|\, (n, \emptyset)$, then $A$ is satisfiable.*

Proof:
Let $\mathcal{T} = \langle W, S', L', R\rangle$ be the loop tree for an expanded tableau for $A \,|\, \emptyset\,;\, [] \,|\, (n, \emptyset)$.

We prove the following proposition: For any prestate $ps = \ldots \,|\, \ldots \,|\, (n, uev)$ in the tableau with $uev \neq \{\mathsf{false}\}$ there is a pre-Hintikka structure $\mathcal{H}_{ps} = \langle S, L\rangle$ for $A$ with $open(S, L) = uev$ . In order to describe the pre-Hintikka structure, we construct a loop tree with one single branch $\pi$ starting with the path in $\mathcal{T}$ that leads to $ps$. By an endless repetition of the loop of this branch we obtain the required $\mathcal{H}_{ps}$ and we obviously have $open(S, L) = open_{\inf}(\pi, L')$.

We proceed by induction on the length $k$ of the longest prestate-branch of the tableau subtree with root $ps$.

$k = 0$: Then $ps$ must be an instance of a terminal rule. If it is an instance of (contr) or (false), then there is nothing to show. If it is an instance of (loop), then $uev$ is $open_{\inf}(\pi', L')$ where $\pi'$ is the corresponding loop in $\mathcal{T}$. The branch of $\mathcal{T}$ that contains $\pi'$ is the required $\pi$.

$k \to k+1$: Assume that the longest branch of the subtree with root $ps$ has length $k+1$. We now make a distinction depending on the type of the rule which has been applied to $ps$ to obtain its son(s). If it is an $\alpha$-rule or a nexttime rule, then the claim follows by the induction hypothesis. Assume that it is a $\beta$-rule:

$$\frac{ps = \beta, \Gamma \mid Ev\,;\; Br \mid (n, uev)}{ps_1 = \beta_1, \Gamma \mid Ev'\,; Br \mid (n_1, uev_1) \qquad ps_2 = \beta_2, \Gamma \mid Ev\,; Br \mid (n_2, uev_2)}\ .$$

Let $m$ be $len(Br) - 1$. If $uev_1$ and $uev_2$ are $\{\mathsf{false}\}$, or if $n_1 > m$ and $n_2 > m$, then also $uev$ is $\{\mathsf{false}\}$, and there is nothing to show.

If $uev_1 = \{\mathsf{false}\}$ (and $uev_2 \neq \{\mathsf{false}\}$), or if $n_1 > m$ , then $uev = uev_2$, and the claim follows by the induction hypothesis. The reversed situation is treated analogously.

There still remains the case in which $uev = uev_1 \cap uev_2$, that is $n_1 \leq m$ and $n_2 \leq m$. Let $s_1$ be the state above $ps$ to which $n_1$ refers, and $n_2$ the state above $ps$ to which $n_2$ refers. By the induction hypothesis there exist two branches $\pi_1$ and $\pi_2$ with the path to $ps$ as common prefix and $open_{\inf}(\pi_1, L') = uev_1$ and $open_{\inf}(\pi_2, L') = uev_2$. These branches can be linearized into a single branch $\pi$ as depicted in Fig. 6.5 (the capital letters denote sections of the path). $L'$ certainly fulfills (PC0-3) and (LC) on $\pi$. We still need to show that $open_{\inf}(\pi, L') = open_{\inf}(\pi_1, L') \cap open_{\inf}(\pi_2, L')$. This, however, is a consequence of Lemma 3.37 since $\pi_1$ and $\pi_2$ are exactly the branches that are visited infinitely many times by $\pi$.

This concludes the proof of the proposition. The theorem follows immediately (by Theorem 3.19) since there exists at least one prestate with $uev = \emptyset$.

□

**4.14 Lemma** *Let $\mathcal{T} = \langle W, S', L', R\rangle$ be the loop tree for an expanded tableau for $A \mid \emptyset \quad [] \mid (n_A, uev_A)$. Then we have: If there is a infinite path $\pi$ through $\mathcal{T}$ with $open(\pi, L) = \emptyset$, and if $L$ fulfills (PC0-3) and (LC) on $\pi$, then $uev_A$ must be $\emptyset$.*

Proof: Let $\pi_1, \ldots, \pi_m$ be the loops of $\mathcal{T}$ that are visited infinitely many times by $\pi$. By Lemma 3.37 we know that $\bigcap_{i=1\ldots m} open_{\inf}(\pi_i, L)$ must be $\emptyset$. Obviously, it is not possible that one loop belongs to an isolated subtree if the other loops do not belong to the same subtree. More precisely, if a subtree at $ws$, $s = \ldots \mid Ev\,;\, Br \mid (n, uev)$, contains some but not all loop states, it cannot be isolated, and therefore $\mathrm{depth}_{\mathcal{T}}(ws)$

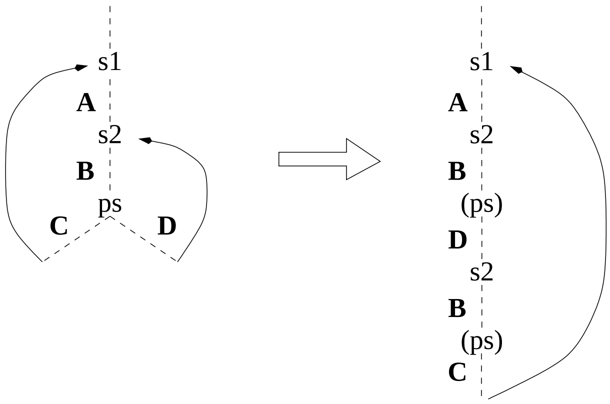



Figure 6.5: Loop linearization.

$= len(Br) > n$ by Lemma 3.32 and Lemma 4.11 b),c). That is, $n \leq m = len(Br) - 1$, and therefore $uev$ is not discarded but consecutively intersected in the $\beta$-rules between $w$ and $ws$.

Let now $w_0 s_0$, $s_0 = \ldots \mid \ldots \mid (n_0, uev_0)$, be the 'deepest' root of a subtree containing all loops $\pi_1, \ldots, \pi_m$, that is, $w_0 s_0$ is also a loop state of at least one $\pi_i$. Note that if branches not visited by $\pi$ loop further back, then the subtree at $w_0 s_0$ is not necessarily isolated.

We prove the following proposition for any $ws \in W$, $s = \Lambda, \mathsf{X}\,\Sigma \mid Ev\,;\, Br \mid (n, uev)$ and any non-empty set $\{i_1, \ldots, i_l\} \subseteq \{1, \ldots, m\}$: If there are loop states $w_{i_1}, \ldots, w_{i_l}$ of the loops $\pi_{i_1}, \ldots, \pi_{i_l}$ below $ws$ (i.e. $ws \leq w_{i_j}$ $(j = 1 \ldots l)$) , then

$$uev \subseteq \bigcap_{j \in \{i_1 \ldots i_l\}} open_{\mathrm{inf}}(\pi_j, L).$$

First we observe that in any instance of (loop) in the tableau $uev = open_{\mathrm{inf}}(\pi', L)$, where $\pi'$ is the corresponding loop of $\mathcal{T}$. Let $j$ be the length of the longest branch of $\mathcal{T}$, and let $i = \mathrm{depth}_{\mathcal{T}}(ws)$. We proceed by induction on $k = j - i$.

$k = 0$: Then $i = j$ and $s$ is is the last state before an instance of (loop). Assume that $ws$ is the loop state of a loop $\pi_i$ $(1 \leq i \leq m)$. By Lemma 4.11 c) and b) we have

$$n \leq \mathrm{depth}_{\mathcal{T}}(ws) = len(Br).$$

We look at the prestates in the tableau between $ws$ and the instance of (loop) which corresponds to $\pi_i$. They must be of the form $\Sigma' \mid Ev'\,;\, Br * ((\Lambda, \mathsf{X}\,\Sigma), Ev) \mid (n', uev')$ with $n' \leq len(Br)$. Then the conditions on the $\beta$-rules ensure $uev' \subseteq open_{\mathrm{inf}}(\pi_i, L)$: we know that $uev' = open_{\mathrm{inf}}(\pi_i, L)$ for the instance of (loop) and for the prestates above we may have intersections with other loops. Therefore, we also have $uev \subseteq open_{\mathrm{inf}}(\pi_i, L)$.

$k \to k + 1$: Then there is at least one successor $wss'$ for which the claim holds by the induction hypothesis and the state $ws$ may again be the last state before an instance of (loop). We look again at the prestates between $ws$ and $wss'$ (resp.

between $ws$ and the instance of (loop)) and by the same arguments as above we get for $uev$ a subset of the intersection of $open_{\inf}(\pi_{i_1}, L), \ldots, open_{\inf}(\pi_{i_l}, L)$.

Since $w_0 s_0$ is above ($\geq$) all loop states of $\pi_1 \ldots \pi_m$ we have for the set $uev_0$ of $s_0$ that $uev_0 \subseteq \bigcap_{i=1\ldots m} open_{\inf}(\pi_i, L) = \emptyset$, that is, $uev_0 = \emptyset$. Using Lemma 4.7 b) we obtain $uev_A = \emptyset$. $\square$

**4.15 Theorem** *(Completeness)*
*If a formula $A$ in negation normal form is satisfiable, then there exists a tableau for $A \mid \emptyset\,;\, [] \mid (n, \emptyset)$ for some $n \in \mathbb{N}$.*

Proof: Assume that $A$ is satisfiable. Then there exists a complete Hintikka structure $\mathcal{H} = \langle S, L\rangle$ for $A$ (see Theorem 3.19). Let $S$ be the sequence $s_0 s_1 \ldots$, and let $\mathcal{T} = \langle W, S', L', R\rangle$ be the loop tree for an expanded tableau for $A \mid \emptyset \;\; [] \mid (n, uev)$. We define inductively a map

$$\varphi : S \to W$$

which provides us with a path $\pi = \varphi(s_0)\varphi(s_1)\ldots$ through $\mathcal{T}$ with the following properties:

(a) $A \in L'(\varphi(s_i))$ if $i = 0$.

(b) $L'(\varphi(s_i)) \subseteq L(s_i)$.

(c) If $C\,\mathcal{U}\,D \in L'(\varphi(s_i))$ and $D \in L(s_i)$, then $D \in L'(\varphi(s_i))$.

$i = 0$: First, $A$ is in $L'(w)$ for every root $w \in W$ since the loop tree stems from a tableau for $A \mid \ldots \mid \ldots$, and $A$ is also in $L(s_0)$, since $\mathcal{H}$ is a Hintikka structure for $A$. Second, there must exist a root $w_0 \in W$ with $L'(w_0) \subseteq L(s_0)$ since in the tableau there is a root state for each possible decomposition of $A$, and $L(s_0)$ must contain at least one set of decomposed formulae ($L(s_0)$ contains $A$ and $L$ fulfills (PC2) and (PC3)). Third, we can choose a $w_0$ so that for each $C\,\mathcal{U}\,D \in L'(w_0)$ with $D \in L(s_0)$ the decomposition $\{C\,\mathcal{U}\,D, D\}$ rather than $\{C\,\mathcal{U}\,D, \mathsf{X}\,(C\,\mathcal{U}\,D)\}$ is a subset of $L'(w_0)$. Set $\varphi(s_0) = w_0$.

$i \to i+1$: Assume that we have defined the map up to $\varphi(s_i)$. We define the sets $next$ and $next'$ as

$$\begin{aligned} next &:= \{C \mid \mathsf{X}\,C \in L(s_i)\} \\ next' &:= \{C \mid \mathsf{X}\,C \in L'(\varphi(s_i))\}. \end{aligned}$$

We know that $next \subseteq L(s_{i+1})$ since $L$ fulfills (LC), and that $next' \subseteq L'(w)$ for every successor $w$ of $\varphi(s_i)$ (see the $(\mathsf{X})$ rule of $\mathsf{PLTL}_T$). Moreover, because of (b), we have $next' \subseteq next$. Again, since in the tableau there is a successor for each possible decomposition of $next'$, and since $L(s_{i+1})$ must contain at least one decomposition, there must exist a $w_{i+1} \in W$ so that (b) and (c) are fulfilled if we set $\varphi(s_{i+1})$ to $w_{i+1}$.

Obviously, $L'$ fulfills (PC0-3) and (LC) on $\pi = \varphi(s_0)\varphi(s_1)\ldots$. Suppose now that there exists an eventuality $C\,\mathcal{U}\,D \in open(\pi, L')$. Then there exists a state $\varphi(s_i)$ so that $C \mathcal{U} D \in L'(\varphi(s_j))$ and $D \notin L'(\varphi(s_j))$ for all $j \geq i$. However, because of (b) and (c) this would mean that $C \mathcal{U} D$ is in $open(S, L)$ as well, which is a contradiction. Applying the previous lemma 4.14 concludes the proof of the theorem. □

## 6.5 Algorithmic Aspects

A decision procedure based on $\mathsf{PLTL}_T$ can execute a depth first search (DFS) starting with the prestate $A \,|\, \emptyset\,;\, []\,|\, Res$, where $A$ is the formula to be checked for satisfiability and *Res* is still unknown (*Res* can be implemented as a reference, just like a Prolog variable). $A$ will subsequently be decomposed according to the rules of $\mathsf{PLTL}_T$ and the *Save* parts keep a record of the current branch. Note that the *Save* parts can be shared among the prestates of a branch. Since the *Res* parts depend on the result parts of the son(s), they will be synthesized in postorder (see also Lemma 4.7 a)), i.e. when returning from the recursion. Clearly, as soon as *Res* has the form $(n, \emptyset)$, we can stop expanding new nodes and report that the formula $A$ is satisfiable (Lemma 4.7 b) and Theorem 4.13).

There are many possible algorithmic improvements. An obvious strategy is to apply the non-branching $\alpha$-rules before the $\beta$-rules. Another idea is to do loop check on prestates rather than states. Imagine that an identical prestate *ps* has been found as in the left diagram of Fig. 6.6. For this situation we obtain by loop check on states the larger tableau of the right diagram (in which only the states are depicted). By a suitable loop check on prestates we need only to expand the left tableau. After an ( $\mathsf{X}$ ) application we apply non-branching rules as long as possible and check then for a loop, i.e. before the first branching. Note that if two states are identical, the prestates that are checked in this way are identical as well. See also Example 4.8.

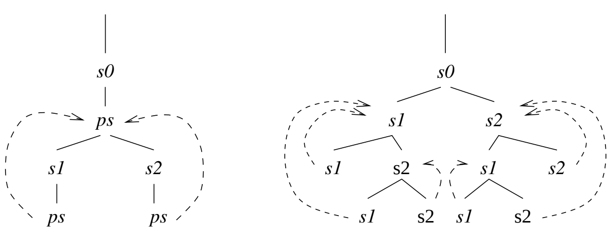



Figure 6.6: Loop check on prestates vs. loop check on states.

In many cases in which the formula is satisfiable we can stop expanding earlier if the synthesis of the result part $(n, uev)$ is treated differently.

Imagine the situation in Fig. 6.7. If we know after the expansion of the first branch of $ps_2$ that the result will be intersected with the result of the first branch of $ps_1$ and that already $uev \cap uev' = \emptyset$, we do not have to expand the second branch of $ps_2$. Since the conditions of the synthesis in the $\beta$-rules depend on the depth of the state $s$ above, it is reasonable to store the intermediate result at $s$ and to update it whenever a new branch has been expanded.

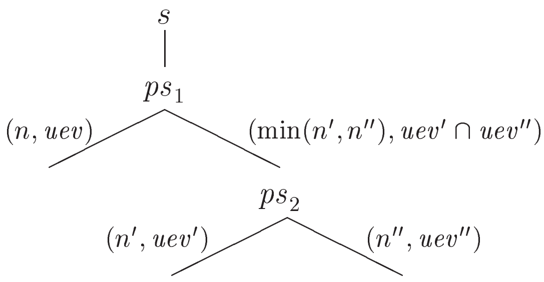



Figure 6.7: Result synthesis.

A nice algorithmic property of the calculus is that we can obtain in a canonical way *linear* models for satisfiable formulae, namely by a linearization of loops as depicted in Fig. 6.5. Of course, such models will in general not be minimal (as the decision procedure does in general not find the shortest way to establish satisfiability).

We conclude this section with a brief discussion of three approaches which may help to prune parts of the tableau.

1. 'Lemma Generation': Based on the classical equivalence

$$(\beta_1 \vee \beta_2) \leftrightarrow (\beta_1 \vee (\beta_2 \wedge \neg\beta_1))$$

we can adapt the $\beta$ rules in the following way:

$$\frac{\beta, \Gamma \mid \ldots \mid (n, uev)}{\beta_1, \Gamma \mid \ldots \mid (n_1, uev_1) \qquad \beta_2, \overline{\beta_1}, \Gamma \mid \ldots \mid (n_2, uev_2)}\ (\beta).$$

The idea is that on the second branch with $\beta_2$ the additional formula $\overline{\beta_1}$ can help to find contradictions earlier, especially if $\overline{\beta_1}$ is 'simple' (e.g. a literal). However, if $\overline{\beta_1}$ contains again $\beta$-formulae, there will also be additional branchings. This adaption of the $\beta$-rules is discussed for classical logic in [17] and for multi-valued logic in [38].

2. 'Use Checking': Assume that in the branching

$$\frac{\beta, \Gamma}{\beta_1, \Gamma \qquad \beta_2, \Gamma}$$

the left branch is expanded first and we observe that $\Gamma$ itself is contradictory, i.e. that we do not 'use' $\beta_1$ in contradictions below. Then we can prune the second branch, since also $\beta_2, \Gamma$ must be contradictory. Use checking is mentioned in the context of a decision procedure for intuitionistic logic [59] and the same idea has been used for classical proof condensation [56, 11].

3. 'Necessary Formulae': In [36] the notion of necessary formulae has been introduced. The basic idea is as follows: let $s$ be a state in the tableau and $\Gamma$ be the set of formulae decomposed between the predecessor and $s$. If we find for some $\beta \in \Gamma$ that both $\beta_1$ and $\beta_2$ are in $\Gamma$, then we can omit this state under certain conditions; it is usually sufficient to satisfy one disjunct.

Another obvious possibility is to store the visited states in a hash-table to avoid recreation of whole subtrees. If there are too many states, one could store only the last $n$ states where $n$ depends on the available memory. Another option is to store only a selection of states, for instance those that belong to an isolated subtree where no path can satisfy all eventualities (i.e. if the root of the subtree has a non-empty *uev* set). In model checking and reachability analysis this approach is also known as 'state-space caching' [45, 33].

# Appendix A

# Implementation Details

## A.1 Introduction

The purpose of this appendix is mainly to ease the future maintenance of the LWB. Of course, not the whole system can be covered; we restrict ourselves to highlighting some 'critical' parts of key components of the LWB. The details of the implementation refer to the LWB version 1.0, which was 'frozen' in May 1996.

### A.1.1 Typography

Within text paragraphs the following typographical conventions are applied:

| | font | example |
|---|---|---|
| variables (C++,LWB) | typewriter | `myvar` |
| literal statements (C++,LWB) | typewriter | `for(int i=0; i<10; i++);` |
| meta-statements | slanted | *loop over A* |
| literal input (LWB) | typewriter | `load(cpc);` |
| meta-input | slanted | *load appropriate module* |
| tools | sans serif | perl |
| filenames | sans serif | foo.C |

Function names (C++, LWB programming language) are always followed by a pair of parentheses, e.g. `f()`, omitting the argument list for brevity. The UNIX-like notation foo.[hC] for files is a short cut for 'foo.h and foo.C'.

C++ code fragments are formatted separately:

```
void my_func ( expr e )
{
  // comment
  for (int i = 1; i ≤ e.nops(); i++ );                // more comment at end of line
}
```

code fragment 1, file: my_file.C

### A.1.2 Unified Modeling Language

For diagrams we use a subset of the Unified Modeling Language (UML), which is an attempt to merge the Booch, the OMT (Object Modeling Technique), and the OOSE (Object-Oriented Software Engineering) methods. The UML prescribes a certain style how to depict information, but in general it does not prescribe what is shown. In fact, most elements of a diagram are optional; they can be there if they help to illustrate the given context, but they do not have to be there. See [15] for a detailed account of the Unified Modeling Language.

#### Class Diagrams

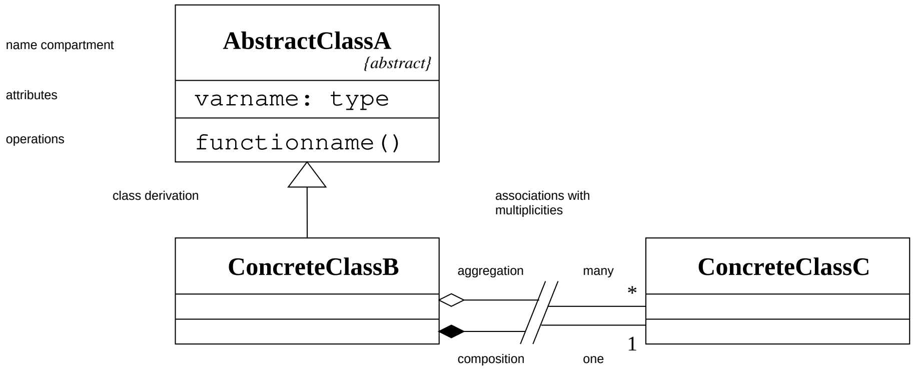



Figure A.1: UML class diagrams.

Instead of marking a class explicitly as abstract, we can also write the class name in italics. Similarly, the names of abstract operations are in italics. The attributes and operations compartments are optional.

The relationship between classes — also called associations — are drawn as a solid path between the classes. An important kind of association is the ‘aggregation’ indicated by diamonds attached to the class which is the aggregate. A filled diamond signifies the strong form of aggregation known as ‘composition’.

#### Object Diagrams

Object diagrams are very much like class diagrams in UML. The name compartment shows the name of the instance and the name of the class, separated by a colon and underlined. If the instance name has no importance, it may be omitted. The optional attributes compartment shows the attribute values.

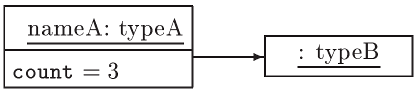



Figure A.2: UML Object diagrams.

### A.1.3 Tools

The following table summarizes the most important tools used for the different parts of the LWB:

| component | tool | version | get version |
|---|---|---|---|
| kernel | C++ | Sun C++4.0.1, SC3.0.1 | `CC -V test.C` |
| modules | C++<br>Prolog | Sun C++4.0.1, SC3.0.1<br>SICStus 2.1 #8 | |
| lexical analyzer | lex | GNU flex version 2.3 | `flex -V` |
| parser | yacc | GNU Bison version 1.21 | `bison -V` |
| user interface<br><br>proof display | X11<br>Motif<br>Tcl/Tk | X11R5<br>OSF/Motif Version 1.2.4<br>tcl7.4,tk4.0 | `strings ‘which mwm‘ \| grep OSF (libraries)` |
| information system | netscape<br>mosaic<br>perl | Netscape 2.02<br><br>perl revision: 4.0.1.8 | `netscape -V`<br><br>`perl -v` |
| project management | make<br>RCS<br>perl<br>sh-scripts | SunOS 5.4 make<br>5.7/5.9<br><br>SunOS 5.4 csh | `(man make)`<br>`(man co/ci)`<br><br>`(man csh)` |

## A.2 The LWB Kernel

### A.2.1 The Expression Handle Class expr

The `expr` class (along with its companion class `exprrep`) is *the* basic object of the LWB kernel. The most relevant part of its definition is presented below.

```
class expr {

  // <friend declarations>

public:
  // constructors, destructors
  expr();                                        // default constructor
  expr(char *s);                                 // a symbol 's' is created
  expr(const expr &f);                           // copy constructor
  ~expr();                                       // destructor

  expr& operator=(const expr &f);                // assignment
  void unassign();                               // unassignment
  expr copy();                                   // explicit copy
  expr subs(const fixedlist &S);                 // S list of substitutions :
```

```
                                        //      S[2] by S[1], S[4] by S[3] ...
  expr evaluate();

  expr& operator[] (int i) const ;                    // subscript [1..nops()]
  expr& index0 () const;                                    // subscript [0]

  int operator==(const expr &f) const;                       // equality test
  int operator≠(const expr &f) const;                      // inequality test
  int identical (const expr &f) const;          // same structure (equal rep's)

  // basic operations
  int type() const;                                // type of toplevel operator
  int nops() const;                 // number of operands of toplevel operator

  // <display functions>

private:
  exprrep *rep;                                  // pointer to actual expression
};
```

code fragment 1, file: expr.h

## A.2.2 The Expression Representation Class `exprrep`

In a few cases `exprrep` provides a default method, e.g. `priority()`. These default methods are virtual and can therefore be overridden by subclasses, but if they are not, the `exprrep` methods are used. This serves also as a technique to lift a feature (function) which is only meaningful for a part of the expression hierarchy to the base class: we then have a (dummy) implementation for all types of expressions, and no downcasting is necessary.

```
//
// class   exprrep   (abstract)
//
//   'Body' class for the class expr. From this class all the types are derived
//   that are intended for use as expressions. It declares the functions to be
//   defined by the derived classes as pure virtual ( virtual f(...) = 0 ).
//

class exprrep {

  // <friend declarations>

public:
  exprrep() : count(0) {};
  virtual ~exprrep() {};

  void ref() { count++; }
  void deref() { count--; }
```

```
  virtual exprrep *copy() = 0;            // creates a copy of *this (not recursive)
  virtual exprrep *subs(const fixedlist &S) = 0;
  virtual exprrep *evaluate() = 0;
  virtual expr& operator[] (int) const = 0;              // subscript [1..nops()]
  virtual expr& index0() const = 0;                                 // subscript [0]

  virtual int hasrep(exprrep *r) = 0;         // returns 1 if r is contained in the
                                             //    expression tree, 0 otherwise

  virtual int type() const = 0;
  virtual int nops() const = 0;

  // <output functions>

  virtual const char* chars() const { return NULL; }
  virtual int priority() const { return maxpriority; }

  // casttype() returns the type to which a type cast can take place
  // it differs from type() for those classes that are derived from a
  // 'generic' base classes (e.g., andop from binop)
  virtual int casttype() const { return type(); }

protected:
  int count;                           // counts pointers to exprrep (not lvalues!)

};
```

code fragment 2, file: exprrep.h

### A.2.3 Programming with Expressions

The following pieces of C++ code illustrate the usage of the `expr` class (see also code fragment 1). The only header file needed is expr.h. After each print statement the corresponding output is inserted. Note that this overview is not complete. A comprehensive collection of `expr` statements can be found in testexpr.C.

As atomic expressions we use three symbols (the constructor of `expr` taking a string creates internally instances of the class `symbol`). Usually, a symboltable manages the symbols.

```
#include "expr.h"

// create (expr-)symbols (not strings)
expr e = "e";
expr f = "f";
expr g = "g";
```

Next we compose the classical formula $(e \to \neg f) \wedge (f \vee (g \leftrightarrow g))$ . The binary and unary functions (`and()`, `imp()`, `not()` etc.) which create operators are also in expr.h.

```
expr s = and( imp(e,not(f)), or(f,eq(g,g)) );
cout << s ;

OUTPUT> (e -> ~f) & (f v (g <-> g))
```

In the following example, we see the basic operations, as for instance type query, number of operands, indexing. The type constant `AND` is defined in globals.h.

```
cout << "type(s) == AND: " << (s.type() == AND)
     << ", nops(s):  " << s.nops() <<"\n";
cout << "s[1] :  " << s[1]
     << " , s[2][2] :  " << s[2][2] << "\n";

OUTPUT> type(s) == AND: 1, nops(s): 2
OUTPUT> s[1] : e -> ~f , s[2][2] : g <-> g
```

Expression can be tested on (syntactical) equality, inequality, and ‘validity’ (which indicates whether the expression is initialized). The structure sharing accelerates the (in)equality tests because first the pointers are compared.

```
expr s2 = s;
expr s3;
cout << "s2 == s :  " << (s2 == s) << "\n"
     << "s != e :  " << (s != e) << "\n"
     << "s3() :  " << s3() << "\n";

OUTPUT> s2 == s : 1
OUTPUT> s != e  : 1
OUTPUT> s3()    : 0
```

The formulae for the non-classical logics can be composed in the same way as classical formulae. E.g. the modal formula $\Box\neg f \wedge \Diamond e$ :

```
s = and ( box(not(f)), dia(e) );
cout << s ;
```

```
OUTPUT> box ~f & dia e
```

The multi-modal formula $\Box_3\neg f \wedge \Diamond_5 e$ :

```
s = and ( box(not(f),3), dia(e,5) );
cout << s ;
```

```
OUTPUT> box3 ~f & dia5 e
```

The tense logic formula $\mathsf{H}(e \wedge \mathsf{P}(\mathsf{G}f \vee \mathsf{F}e))$ :

```
s = tH( and(e,tP( or( tG(f), tF(e) ))) );
cout << s ;
```

```
OUTPUT> H (e & P (G f v F e))
```

The linear logic formula $e \otimes (f \oplus (g \wp g)) \multimap e \otimes f \oplus f \otimes g$ :

```
s = lleq( times(e,llplus(f,par(g,g))), llplus(times(e,f),times(f,g)) );
cout << s ;
```

```
OUTPUT> e X (f + (g | g)) o-o e X f + f X g
```

Strings and integers can be converted to expressions. If an expression has the corresponding type (the programmer must ensure this!), the other direction is possible, too. The conversion functions are also in expr.h.

```
expr n = int2expr( 233 );
expr m = string2expr( "hello" );
int i = expr2int(n);
const char *str = expr2string(m);
```

A typical function which traverses a formula is the length of a classical formula. All non-classical parts (if there are any in the argument) are treated as atoms (`default` case of the `switch` statement).

```
//
// cpc_length - length of a classical formula
//

int cpc_length(const expr &A)
{
  switch ( A.type() ) {
  case OR :
  case AND :
  case IMP :
  case EQ :
    { return cpc_length(A[1]) + cpc_length(A[2]) + 1; }
```

```
    case NOT :
      { return cpc_length(A[1]) + 1; }
    default :
      { return 1; }
    }  // switch
}  // cpc_length
```

code fragment 3, file: cpc_general.C

See also page 22 ‘Passing Expressions as Arguments’.

### Compiling and Linking

For compiling the include-path to expr.h is needed. For linking the kernel library liblwbkernel.so and an implementation of the global output devices (in plainversion.o) is basically needed.

Assume that statements as above are in the file exprdemo.C. Then the minimal compile and link command is

```
CC -g -I$(LWBHOME)/main/include exprdemo.C  -o exprdemo
      -L$(LWBHOME)/lib -llwbkernel $(LWBHOME)/lib/plainversion.o
      -L/usr/ccs/lib -ltermlib
```

where `$(LWBHOME)` is the directory in which the LWB 1.0 is installed.

### File Overview

#### `expr` files

| | |
|---|---|
| expr.[hC] : | handle class for LWB expressions. |
| exprrep.[hC] : | body class for LWB expressions (internal). |
| exprstruct.[hC] : | `exprrep` structure classes, e.g., lists and function expressions. |
| globals.[hC] : | global variables of the LWB kernel, e.g., type constants, operator priorities, operator symbols. |
| testexpr.C : | test file for the `expr` and `exprrep` classes. This file can also be accessed as html-document. |
| teststruct.C : | test file for the `exprstruct` classes. |

#### LWB output

| | |
|---|---|
| device.[hC] : | stream-like device class. |
| plainversion.C : | declarations for the plain ASCII version, especially of the global output devices. |

#### kernel library

| | |
|---|---|
| liblwbkernel.so : | shared library, combining expr.o, exprrep.o, exprstruct.o and a few utility files. |

### A.2.4 LWB Output

```
//
// (abstract) class  device
//
//   Intermediate class to hand over output (mainly strings) either to
//   real streams like cout or a file (derived class osdevice) or
//   a user interface routine that accepts strings (derived class to be
//   implemented by the user interface).
//   The main virtual function is  print  (for strings) that must be provided
//   by all derived classes.


class device;

typedef device& (*devicemanip)(device&);          // device manipulator type

class device {
public:
  virtual void flush() {};
  virtual void print(const char* s) = 0;

  // (indent functions)

  device& operator <<(const char* s) { print(s); return *this; }
  device& operator <<(const char c);
  device& operator <<(const double d);
  device& operator <<(const int i);
  device& operator <<(const expr &e);
  device& operator <<(const exprrep &r);
  device& operator <<(const devicemanip m) { return m(*this); }
  device& printf( char *format, ... );

  int indentation;

private:
  static const int maxindent;
};
```

code fragment 4, file: device.h

Different classes can be derived from `device` for the different purposes, e.g., writing to streams (class `osdevice`, code fragment 5). The two most frequently used `ostream` manipulators, `endl` and `flush`, are also provided for `device` (not only `osdevice`).

```
//
// class  osdevice
//   Output stream device: is initialized with an  ostream-reference to which
//   the calls are forwarded.
```

```
class osdevice : public device {
public:
  osdevice(ostream &initostr) : ostr(initostr) { assert(!!ostr);};

  void flush() { ostr.flush(); }
  void print(const char* s) { ostr << s; }
private:
  ostream& ostr;                                  // only reference!
};
```

code fragment 5, file: device.h

### Example of Algorithmic Information

The following example shows the skeleton of a typical information management. Note that the indentation features are available for any device (not just `infodev`).

```
void myalgo() {
  info( ilev::algo, "myalgo" );
  ...
}

void mysubalgo() {
  infodev.incindent();          // (optional setting of the indentation)
  info( ilev::subalgo, "mysubalgo" );
  ...
  infodev.decindent();
}


// if the output has to be prepared, or not only strings are
// to be displayed :

if ( getinfolevel() >= ilev::where ) {
  ...                                // computations for the following output
  infodev << ... << endl;                          // (without indentation)
  infodev << indent << ... << endl;                   // (with indentation)
  infodev.printindented("line1\nline2");   // (several indented lines at once)
}

// setting the infolevel (usually chosen by the user)
{
  setinfolevel( ilev::where );
}
```

code fragment 6, file:

All shown features are part of the LWB kernel and are automatically available when expr.h is included.

### A.2.5 Functions and Function Descriptors

#### The Function Class `function`

```
exprrep
   |
function
```

A function expression consists of a functor and an argument list. Within a formula, it can be viewed as an operator with arbitrary arity. The main difference to other expressions is that it may later evaluate to the result of a function call.

```
//
// class function : functional expressions ( functor(arglist) )
//
//   If a function is evaluated (member function evaluate) it is checked if
//   the functor contains a function descriptor (class funcdescr), and if so
//   the descriptor is used to call an actual function with arglist as
//   argument (funcdescr::apply).

class function : public exprrep {
public:
  function(const expr &e, const expr &L);
  ...
  exprrep *evaluate();
  ...
private:
  expr functor;
  fixedlist *arglist;
};
```
code fragment 7, file: exprstruct.h

The `evaluate()` method is quite different from evaluation methods of other expression subclasses. Provided that the functor evaluates to a function descriptor, the result of the function that the descriptor contains is returned. In this case, the evaluated argument list has to be deleted if it is temporary (reference count 0), that is, if the evaluated list differs from the original list. Since the result might be a subformula of an argument, it has to be protected from this deletion by temporarily increasing the reference count.

```
exprrep* function::evaluate() {
  exprrep *res1 = functor.rep->evaluate();
  fixedlist *res2 = (fixedlist*) arglist->evaluate();
  if ( res1->casttype() == FUNCDESCR ) {
    exprrep *res = ((funcdescr *) res1)->apply(res2);
    if (res2->count == 0) {
      // delete the newly created, evaluated arglist, but protect
      // the result which could be a part of an argument from deletion
```

```
      res→ref(); delete res2; res→deref();
    }
    return res;
  }
  else
    return
      (functor.rep==res1 && arglist==res2) ? this : new function(res1,res2);
}
```

code fragment 8, file: exprstruct.C

Note that `function::evaluate()` is only called if an existing functional expression is evaluated. If it does not exist yet, the interpreter calls the `apply()` method directly (primitive `invoke`), without creating a function expression first.

Remark: `arglist` is declared directly as a pointer to a `fixedlist` (and not as an `expr` containing a `fixedlist`) to facilitate the access to the list methods, that is, to save conversions. The function constructor and destructor must therefore take care of its reference count.

### The Function Descriptor Class `funcdescr`

```
  exprrep
     |
 funcdescr
```

The function descriptors provide the connection to executable functions. Since more than just the connection to C functions is offered, a function pointer is not sufficient. The abstract class `funcdescr` prescribes that derived classes must supply a function `apply()`, which is supposed to call a real function with the arguments in `*args`. `args` will be a pointer to the evaluated argument list.

```
//
// (abstract) class funcdescr : description of 'real' functions
//
//   Serves as an expr function pointer. The only member function
//   not available through exprrep is apply, which is supposed to
//   execute a real function call. See also class function.

class funcdescr : public exprrep {
public:
  ...
  virtual exprrep *apply(fixedlist *args) = 0;
  ...
};
```

code fragment 9, file: exprstruct.h

### The C Function Descriptor Class Cfuncdescr

```
  exprrep
     |
 funcdescr
     |
Cfuncdescr
```

The descriptor for C functions consists of a C function pointer which is called by the `apply()` member function.

```
//
// class Cfuncdescr : C function pointer
//

typedef expr (*Cfuncptr) (const fixedlist &);

class Cfuncdescr : public funcdescr {
public:
   Cfuncdescr(Cfuncptr iptr) : funcptr(iptr), s(NULL) {};
   ...
   exprrep *apply(fixedlist *args);
   ...
private:
   Cfuncptr funcptr;
   symbol *s;
};


// ————————————————————————————————
// class Cfuncdescr implementation
//

exprrep *Cfuncdescr::apply(fixedlist *args) {
   expr e = funcptr(*args);                    // (funcptr returns an expr)

   // Prevent deletion of object by the destructor of e (in case that
   // e is the only reference).
   exprrep *rep = e.rep;
   rep->deref();
   e.rep = NULL;
   return rep;
}
```

code fragment 10, file: mach.[hC]

Remarks:

- The C function returns an `expr`, while `apply()` as part of the evaluation process has to return an `exprrep`. So one has to ensure that the destructor of the result does not delete the result.

- The symbol `s` of `Cfuncdescr` is used to identify the function name when an error (e.g. argument check) is raised. It is set by an install method of the symboltable and is queried by `Cfuncerror()` in mach.C.

### The Prolog Function Descriptor Class `SPfuncdescr`

```
  exprrep
     |
 funcdescr
     |
SPfuncdescr
```

In the following we present an exemplary way how third party systems can be embedded into the LWB.

1. A new function descriptor derived from `funcdescr` must be defined which holds the information needed to do an external function call.

2. Two conversion functions, LWB expression to foreign expression and foreign expression to LWB expression, must be written.

The class `SPfuncdescr` provides a connection to SICStus Prolog (SP). The SICStus Prolog's runtime library can be linked to the LWB, so that predicates from compiled SP files (extension .ql) can be accessed.

```
//
// class   SPfuncdescr
//
//    Funcdescr class for SP predicates. The predicates must have a 'functional'
//    shape, i.e. the last argument must be a SP variable where the result of a
//    query will be put. 'nargs' denotes the number of arguments of such a
//    function and is therefore predicate arity minus one.
//

class SPfuncdescr : public funcdescr {
public:
  SPfuncdescr(char *ipredname, int inargs, char *imodulename);
  ...
  exprrep *apply(fixedlist *args) ;
  ...
  int nargs;
private:
  SP_pred_ref pred;
  char *predname;
  char *modulename;
};
```

code fragment 11, file: SPfuncdescr.h

The `apply()` method of a 'foreign' function descriptor is straightforward:

1. Convert the `expr` arguments to the foreign structures.
2. Call the foreign function.
3. Convert the foreign result back to an `expr`.

```
exprrep* SPfuncdescr::apply(fixedlist *args) {
  if ( nargs ≠ args→nops() ) {
    warning("wrong number of arguments for SICStus predicate");
    return NULLexpr.getrep();
  }
  SP_term SPargs[SPmaxargs];
  for (int i=0; i < args→nops(); i++)                    // convert arguments
    SPargs[i] = SPexpr( (*args)[i+1] );
  SP_put_variable(&SPargs[nargs]);                // 0..n-1 : args,   n : result
  if ( !pred ) {
    pred = SP_predicate(predname, nargs+1, modulename);
    if ( !pred ) {
      warning("predicate not found");
      return NULLexpr.getrep();
    }
  }
  i=SP_query(pred,
             &SPargs[0], &SPargs[1], &SPargs[2], &SPargs[3], &SPargs[4],
             &SPargs[5], &SPargs[6], &SPargs[7], &SPargs[8], &SPargs[9]);
  return Cexprrep( SPargs[nargs] );                        // reconvert result
}
```
code fragment 12, file: SPfuncdescr.C

Note that it is possible to declare instances of `SPfuncdescr` that do not correspond to a SP predicate. Only when the function is called the first time, the binding with `SP_predicate(...)` is attempted (in `SPfuncdescr::apply()`).

The following table lists the most important SP functions of the SP runtime library and how they are called (listed in the order of use). See also [26] (chapter 7) or [25] (chapter 8).

| SP function | called from LWB function |
|---|---|
| `SP_initialize()` | `SPcheckinit()` |
| `SP_load( char* filename)` | `SP_I_load`, interface function that the user can access by `> SP_load("filename");` |
| `SP_put_compound(...)`<br>`SP_put_arg(...)` | `SPexpr`, converts an `expr` in a `SP_term` |
| `SP_put_variable`<br>`SP_predicate(...)`<br>`SP_query(...)` | `SPfuncdescr::apply` |
| `SP_get_compound(...)`<br>`SP_get_arg(...)` | `Cexprrep`, converts a `SP_term` in an `expr` |

The conversions have the following shape:

```
//
// SPexpr :     expr -> SP_term
//

static SP_term SPexpr(const expr &e) {
  SP_term res;
  switch ( e.getrep()→casttype() ) {
  case SYMBOL: {
    SP_put_string(&res, (char*)(e.chars()));
    return res;
  }
  case BINOP : {
    SP_put_compound(&res, (char*) e.chars(), 2);
    SP_put_arg(1, res, SPexpr(e[1]));
    SP_put_arg(2, res, SPexpr(e[2]));
    return res;
  }
  case UNOP: {
    SP_put_compound(&res, (char*) e.chars(), 1);
    SP_put_arg(1, res, SPexpr(e[1]));
    return res;
  }
  ...
}

// ————————————————————————————————
// Cexpr :     SP_term -> expr
//

static expr Cexpr (const SP_term &t)
{ return expr(Cexprrep(t)); }

static exprrep *Cexprrep (const SP_term &t) {
  SP_term arg1,arg2;
  char *functor,*atom;
  int arity;
  if (SP_is_compound(t)) {                    //  Tests whether t is non-atom
    SP_get_compound(t, &functor, &arity);               // get connective
    ...
    if (strcmp(functor,andop::descr.chars)==0) {
      SP_get_arg(1, t, &arg1);
      SP_get_arg(2, t, &arg2);
      return new andop( Cexprrep(arg1), Cexprrep(arg2) );
    };
    if (strcmp(functor,notop::descr.chars)==0) {
      SP_get_arg(1, t, &arg1);
      return new notop( Cexprrep(arg1) );
    }
    ...
```

```
    }
    else {                                              // no compound term
      if ( SP_is_atom(t) ) {
        SP_get_string(t, &atom);
        if ( strcmp("[]",atom) == 0 )
          return new fixedlist(0);
        else
          return ms.whichmodule(atom)→st.lookup(atom);
      }
      ...
}
```
code fragment 13, file: SPinteract.C

**Limitations/Requirements of `SPfuncdescr`**

- The Prolog files to be loaded must be in compiled form (extension '.ql').
- The last argument of the predicate must be a SP variable which holds the result at the end of a query.
- Currently, `SPfuncdescr` allows only one arity per predicate name.
- The arity may not exceed `SPmaxargs` (defined in SPinteract.C). Exactly `SPmaxargs` arguments are passed to the SP function `SP_query(...)`.

### A.2.6 Expression Structures

**The Array Class `fixedlist`**

The class `fixedlist` defines a list of expressions with fixed size and is therefore implemented as an array. At the user level '`array`' is also the keyword to construct such fixed lists.

There are a few list specific commands available, as for example `member()` and `append()`. If a list is mostly used as an array with few changes of the size, `fixedlist` is appropriate. For instance, the argument list of a function expression (class `function`) is implemented as `fixedlist`. However, if frequent resizing is required, the `linkedlist` implementation is better (see also section 3.4.6). Both versions have constructors that accept the other list type for initialization.

**The Class Definition**

```
 exprrep
    |
fixedlist
```

`fixedlist` provides some convenience constructors which allow starting off with 1, 2, and 3 expressions. Note the difference between the copy constructor `fixedlist (const fixedlist &f)` and the `copy()` method required by `exprrep`. While the copy constructor initializes the current `fixedlist` with an existing one, the copy method returns an identical copy of the `fixedlist`. In both cases only the actual `fixedlist` data, that is, the `expr` array, is copied; the entries, however, are shared.

```
//
// class fixedlist : dynamic array of expr's
//
//   It may be used either as an expr object (exprrep respectively)
//   or as a class of its own. Only in the latter case the list specific
//   functions are accessible. For uniformity the same list functions are
//   provided as in class linkedlist. Functions affecting the size of the
//   array are more expensive, though.

class fixedlist : public exprrep {
  // (friend declarations)
public:
  fixedlist(): size(0), op(NULL) {};                              // empty list
  fixedlist(expr iop[], const short isize);              // init by array of expr
  fixedlist(const expr &e1);                                  //     by single expr
  fixedlist(const expr &e1, const expr &e2);                    //     by two expr
  fixedlist(const expr &e1, const expr &e2, const expr &e3);  // by three expr
  fixedlist(const fixedlist &f);                      //     by other fixed list
  fixedlist(const linkedlist &l);                            //     by linked list
  fixedlist(const short isize);                          //     by size definition
  ~fixedlist();

  // (methods required by exprrep, e.g. operator [])
  expr& operator[] (int i) const { return op[i-1]; }
  ...

  // list specific functions (not available through exprrep pointers)
  fixedlist& operator=(const fixedlist &f);                       // assignment
  fixedlist& append(const expr &e);
  fixedlist& append(const fixedlist& f);             // copy f and append to *this
  fixedlist& prepend(const expr &e);
  fixedlist& prepend(const fixedlist& f);           // copy f and prepend to *this
  fixedlist& push(const expr &e) { return prepend(e); }
  expr pop();

  fixedlist& sort(exprcmpfunc exprcmp);
  int member(const expr &e) const ;
  int empty() const { return size==0; }

  int size;
private:
  ...
  expr *op;
};
```

code fragment 14, file: exprstruct.h

### The List Class `linkedlist`

`linkedlist` should be used if many prepend and append operations are executed. Since pointers are kept for the beginning and the end of the list, both prepend and append are efficient. Direct access to list members via the subscript operator, however, is not efficient because it requires each time a (partial) scan.

### The Class Definition

```
  exprrep
     |
linkedlist
```

```
//
// class  linkedlist :   Simple linked list of expr's.
//
//   It may be used either as an  expr  object (exprrep respectively)
//   or as a class of its own. Only in the latter case the list specific
//   functions are accessible.

class linkedlist : public exprrep {
public:
  linkedlist() : first(NULL), last(NULL), size(0) {};              // empty list
  linkedlist(const linkedlist &l);                         // init by other linked list
  linkedlist(const fixedlist &f);                              //        by fixed list
  linkedlist(const expr &e);                                  //        by single expr
  ~linkedlist();
  ...

  // list specific functions (not available through exprrep pointers)
  linkedlist& operator=(const linkedlist &l);                        // assignment
  linkedlist& append(const expr &e);
  linkedlist& append(const linkedlist& l);           // copy l and append to *this
  linkedlist& prepend(const expr &e) { return push(e); }
  linkedlist& prepend(const linkedlist& l);          // copy l and prepend to *this
  linkedlist& push(const expr &e);
  expr pop();
  linkedlist& delfirst(const expr &e);              // delete first occurrence of e
  linkedlist& replacefirst(const expr &e, const expr &e1);
  ...
  int member(const expr &e) const ;
  int empty() const { return first==NULL; }

  listnode *first;
  listnode *last;
  int size;

private:
  ...
};
```

code fragment 15, file: exprstruct.h

### The Class `listnode`

The `listnode` class provides its own memory management by overloading the `new()` and `delete()` operator. The items are allocated in chunks instead of one by one. The speed-up in algorithms with frequent `push()`'s and `pop()`'s is considerable. Once allocated chunks are reserved for the `listnode` class and are never freed for general usage again. If a `listnode` is deleted, it is simply prepended to the list of free nodes. The memory reservation is acceptable because the `listnode` size is very small (two pointers). The implementation follows closely the example in [51, 6.3].

### Using Lists as Expressions

First of all, both list types can be used just as lists of expressions whenever a collection of expressions is needed. However, if a list is extracted from an `expr`, and conversely, if a list is passed back as as an `expr`, some rules must be observed.

If an LWB expression represents a list, it can be accessed using the convenience functions `expr2fixedlist()` and `expr2linkedlist()` (in expr.h) which return a pointer to a constant `fixedlist` and `linkedlist`, respectively. In fact, they do a simple type cast which explains why the objects may not be changed. Changes are only allowed on copies. Do *never* pass addresses of local list instances out to an `expr`: when the scope is left, the list is destroyed and the `expr` would contain a dangling reference with unpredictable consequences. Addresses of local variables should in general not be passed outside their scopes.

---

```
expr convert( expr e ) {
  {
    if ( e.type() == FIXEDLIST ) {
      const fixedlist *fl = expr2fixedlist(e);
      // only the const member functions are allowed
      fl→member(b);                                  // allowed
      fl→pop();                     // not allowed by the compiler

      // we take a copy and change it
      fixedlist *flcopy = new fixedlist(*fl);
      flcopy→pop();
      return expr( flcopy );
    }
    else return e;
  }
}


// DON'T

expr desaster () {
```

```
  fixedlist f(...);
  ...
  return expr( &f );
}


// correct

expr okay () {
  fixedlist *f = new fixedlist(...);
  ...
  return expr( f );
}
```

code fragment 16, file:

### The Stack Class `stack`

```
//
// class   stack   :   Reference counted stack of expr's.
//
//   As additional node data an integer is available. If the integer is not
//   used, it behaves like an ordinary stack.
//   The reference counting allows the memory sharing of whole substacks.
//   The member functions push and pop and the copy constructor take care of
//   the proper count adaptions.


class stack {

  friend device& operator << (device &dev,const stack &s);

public:
  stack() : top(NULL) {};                                    // empty stack
  stack(const stack &s);                              // init. by other stack
  stack(const expr &e, const int i=0);                //        by first expr
  ~stack();

  stack& operator=(const stack &s);                           // assignment

  stack& push(const expr &e, const int info=0);
  stack& notmempush(const expr &e,const int info=0 )
  { return member(e) ? *this : push(e,info); }
  expr pop();
  expr pop(int &i);                                 // returns int info in i

  int member(const expr &e) const ;
  int member(const expr &e, int &i) const ;            // returns info in i
  int empty() const { return top==NULL; }
  int length() const;

  stacknode *top;
```

```
};
```

code fragment 17, file: stack.h

### Programming with Structures

The following pieces of C++ code illustrate the usage of the `fixedlist`, `linkedlist`, and `stack` class. The only header files needed are expr.h and exprstruct.h for the lists and stack.h for the stack.

After each print statement the corresponding output is inserted. Note that this overview is not complete. A comprehensive collection of list statements can be found in teststruct.C. For the basic `expr` programming see Sect. A.2.3. The minimal compile and link command can also be found there (cf p. 184) including a file overview.

An example how the stack class can be used in a decision procedure is in cpc_provability.C.

As atomic expressions we use two symbols (the constructor of `expr` taking a string creates internally instances of the class `symbol`). Usually, symbols would be inserted in a symboltable.

```
#include "expr.h"
#include "exprstruct.h"
#include "stack.h"

expr a = "a";
expr b = "b";
```

First we create a two element `fixedlist`. To display a list, the method `ascii()` (required by `exprrep`) is used, which takes the global device `dev` as argument here.

```
fixedlist fl(2);                              // alternative: fl(a,not(b))
fl[1] = a;
fl[2] = not(b);
fl.ascii(dev); dev << "\t// fixedlist\n";
OUTPUT> [a, ~b] // fixedlist
```

Linked lists are constructed element-wise.

```
// creating a linkedlist
linkedlist ll(a);
ll.append(not(b));
ll.pop();
ll.append(and(a,a)).prepend(or(b,b));
ll.ascii(dev); dev << "\t// linkedlist\n";
OUTPUT> [b v b, ~b, a & a]      // linkedlist
```

We create the shared stack structure of Fig. 2.16. The integer arguments to the `push()` function are optional.

```
stack s1(a,1);                          // init by first expression
s1.push(a,2).push(a,3);

stack s2(s1);                           // init 2nd stack by first
s1.push(a,4).push(a,5);
s2.push(b,6).push(b,7);

stack s3(s2);                           // init 3rd stack by 2nd
s2.push(b,8);
s3.push(a,9);

dev << "s1 :  " << s1 << "\n";
dev << "s2 :  " << s2 << "\n";
dev << "s3 :  " << s3 << "\n";
OUTPUT> s1 :  [a, a, a, a, a]
OUTPUT> s2 :  [b, b, b, a, a, a]
OUTPUT> s3 :  [a, b, b, a, a, a]
```

To iterate over a `fixedlist`, we use a common `for` loop and over a `linkedlist` or a `stack` a `for` loop with an iterator (defined in exprstruct.h and stack.h, respectively).

```
// iterating over fixedlists
for ( int i = 1; i <= fl.size; i++ )
   fl[i];

// iterating over linkedlists
for (listiterator it(ll); it.valid(); it.next())
   it.get();

// iterating over stacks
for (stackiterator it(s3); it.valid(); it.next())
   it.get();
```

# A.3 Lexical Analysis and Parsing

## A.3.1 Lexical Analysis

### General Remarks

- The function names, macros, and variables starting with the prefix `yy` or `YY` are all provided/defined by lex or by yacc.
- Many variables in the different code fragments are used for error localization, e.g. `tokenpos`, `savebuf` etc. See page 203 for explanations.

### Input from Strings

The function `int yyparse(char* s)`, which is defined in the definition section of lex.l, initializes variables and toggles to prepare string scanning and calls then the standard (yacc-defined) function `int yyparse()`.

```
static char *myinputlim;      /* points to char that follows the parsed string */
static char *myinputptr;                     /* current position in parsed string */

static int my_yyinput(char *buf, int max_size) {
  int n = min(max_size, myinputlim-myinputptr);
  if (n > 0) {
    memcpy(buf, myinputptr, n);
    myinputptr += n;
  }
  return n;
}


//
// int yyparse(char* s):
//   Global function provided for calls from outside.
//   It puts the parser in a state in which the string  s  can be parsed.
//

int yyparse(char* s) {
  myinputlim = s+strlen(s);
  myinputptr = s;
  linestart = s; tokenpos = 0; strtokenpos = 0; yylineno = 1;
  readstate = readingstring;
  int i = yyparse();
  readstate = readingstdin;
  yyrestart(yyin);             // specific to flex, arranges to start reading this file
  return i;
}
```
code fragment 18, file: lex.l

The read macro `YY_INPUT` (see also code fragment 21) redirects its arguments to `int my_yyinput(char *buf, int max_size)`, which fills at most `max_size` characters in `buf`. The static variables `myinputptr` points to the first unread character of the string (in case that the string is longer than `max_size`).

### Input from Files

The functions `readfromfile()` and `yywrap()` are defined in the user section of lex.l.

```
//
// void readfromfile(const char *name) :
//   Global function, provided for calls from outside. It switches the lexer
//   to a state, in which the named file is scanned. At the end of the file
//   yywrap resets to the state, where it left off.
//
```

```
//   Specific to flex are :
//      YY_BUFFER_STATE    :  flex buffer type
//      YY_CURRENT_BUFFER :  current buffer
//      yy_create_buffer ( FILE*, sizez0 )
//      yy_switch_to_buffer( flexbuf );
//      yy_delete_buffer( flexbuf );
//

#include <sys/stat.h>

extern void parsewarning(const char *where, const char *s);

void readfromfile(const char *name) {
  FILE* newfile = fopen( name, "r" );
  if ( !newfile ) {
    parsewarning("read","file opening failed");
    return;
  }
  else {
    // (check if directory ...)
  }
  if (!linestart) {
    linestart = savebuf;
    savebuf[0] = NULL;
  }
  parserstate.save();
  infile = newfile;
  infilename = strdup(name);
  yylineno = 1;
  readstate = readingfile;
  YY_BUFFER_STATE infilebuffer = yy_create_buffer ( infile, YY_BUF_SIZE );
  yy_switch_to_buffer( infilebuffer );
  if ( getinfolevel() >= ilev::algo )
    infodev << "Parsing file \"" << name << "\":" << endl;
}
```

code fragment 19, file: lex.l

The counterpart of `readfromfile()` is the function `yywrap()`. It is automatically called when the end of the current input file is reached.

```
//
// int yywrap() :
//   When the lexer encounters an end of file, this routine is called to find
//   out what to do next. If it returns 0, the scanner continues scanning,
//   while if it returns 1, the scanner returns a zero token to report the
//   end of file.
//

int yywrap() {
  if ( !parserstate.isempty() ) {
```

```
      parserstate.restore();
      if ( getinfolevel() ≥ ilev::where )
        if (infilename)
          infodev << "Returning to file \"" << infilename << "\"." << endl;
        else
          infodev << "Returning to toplevel parser state." << endl;
      return 0;
    }
    else
      return 1;
}
```

code fragment 20, file: lex.l

### The Central Input Macro YY_INPUT()

`YY_INPUT()`, a macro specific to flex, is used to read blocks of data. It is called as:

```
YY_INPUT( buffer, result, max_size );
```

where `buffer` is a character buffer, `result` will be the number of read characters, and `max_size` is the size of the buffer. The predefined macro has been adapted, mainly to set the prompt at the right time and to save the input line for possible error messages (variable `savebuf`).

```
/*
 * YY_INPUT (buf,result,max_size) :
 *   (flex speciality, taken from lex.C and adapted )
 *   saves the buffer into the (static) variable  savebuf  which is used
 *   for parse error messages.
*/

#undef YY_INPUT
#define YY_INPUT(buf,result,max_size) \
        if ( readstate == readingstdin ) \
          setprompt();\
        if (readstate == readingstring)\
          result = my_yyinput( (char *) buf, max_size );\
        else { \
          if ( (result = read( fileno(yyin), (char *) buf, max_size )) < 0 ) \
            YY_FATAL_ERROR( "read() in flex scanner failed" );\
          if ( result > 0 ) {\
            int n = result ≤ maxsavebuf ? result-1 : maxsavebuf-1; \
            strncpy( savebuf, buf, n ); \
            savebuf[n] = NULL; \
            linestart = savebuf; tokenpos = 0; \
          } \
        }
```

code fragment 21, file: lex.l

The two local toggle variables, `readingafile` and `readingastring`, indicate whether the parser is about to read a file or a string. If none is true, which is the default, the input is taken from the default value of `yyin`, which is stdin. Since the input from strings and files does not appear on the screen, no prompt is written.

### Parse Error Localization

The following variables are used in order to keep track of the current position:

```
static char  savebuf[maxsavebuf];
char* linestart;
int  yylineno;
int  tokenpos;
int  strtokenpos;
```

`savebuf` is a copy of the currently parsed block of characters. When scanning strings or files, this block can contain several input lines.

`linestart` points to the beginning of the current line within `savebuf`. When a switch to a file takes place, not the whole `savebuf` is saved, but only the rest of the block starting at `linestart` (`parserstatestack::save` in lex.l). When returning from a file, it is copied back to the beginning of `savebuf` and `linestart` is reset accordingly (`parserstatestack::restore`).

`yylineno` stores the number of the current line within the file and within the string, respectively.

`tokenpos` stores the character position of the current token within the current line.

`strtokenpos` stores the character position of `*linestart` within the *string* when scanning strings. This position is needed by the GUI for which the `linestart` pointer is useless since it points to a copy and not the input region itself. When reading from stdin, `strtokenpos` is always 0.

These variables are all part of the lexer state and must therefore be saved and restored when switching between different sources. When a parse error occurs, they are used in order to set up the arguments of `showparseerror()`, a function which actually displays the error and which hence has to be implemented twice; once for the LWB (file plainversion.C) and once for the XLWB (file parserconnection.C). Another variable which is used for `showparseerror()` is the length of the current token, `yyleng`, which is automatically updated by lex (there is always exactly one current token, so it need not to be stored).

## A.3.2 Parsing

### The Yacc Error Routine

The yacc error routine `yyerror()` is either called automatically (syntax errors) or explicitly before the macro `YYERROR` (context-sensitive) errors. Its task is also to display the erroneous token (internal LWB function `showparseerror()`).

```
//
// void yyerror(const char* s) :
//    Automatically called function in case of a yacc syntax error.
//

extern void reset_code_generator_and_scope();                    // in gen.C

void yyerror(const char* s) {

  ...
  warndev << s;
  ...
  warndev << " in line " << yylineno;
  warndev << " at '" << yytext << "':\n";
  ...
  showparseerror( linestart,
                  tokenpos-yyleng+1, strtokenpos+tokenpos-yyleng+1, yyleng );
  warndev.flush();
  isrecovering = 1;

  reset_code_generator_and_scope();
}
```

code fragment 22, file: lex.l

Since internal lex variables are used for the error localization (cf p. 203), the function is defined in the user code section of the lex input file lex.l.

**File Overview**

**lex files**

| | |
|---|---|
| lex.l : | lex definition file (input for lex). |
| lex.C : | lex output file, C(++)–Code. |

**yacc files**

| | |
|---|---|
| yacc.y : | yacc definition file (input for yacc). |
| yacctions.C : | include file for yacc.y containing C++ code for the more complex actions of different rules. |
| yacc.C : | yacc output file, C(++)–Code. (original: y.tab.c) |
| yacc.C.h : | additional yacc output file, C(++)–Code, contains the token `#define`'s used in lex.l . (original: y.tab.h) |
| yacc.C.output : | extra output file containing verbose descriptions of the parser states and what is done for each type of look-ahead token in that state. This file also describes all the conflicts. (original: y.output) |
| liblwbparser.so : | shared library, combining yacc.o and lex.o (beside others). |

Remark: only the output files of lex and yacc are intended for distribution. Therefore, the usage of special lex and yacc implementations is unproblematic.

### A.3.3 The Interpreter

Here we make a few remarks regarding the LWB Abstract Machine. They should help to find your way through the implementation.

#### LWB Abstract Machine

The *LWB Abstract Machine* is implemented as the C++ class `lwbAM` in the files mach.h[C]. It contains of course more than just an instruction pointer: for instance, a pointer to the current activation record (also called frame) on the stack, and a pointer to the saved machine status. See also [1], Sect. 5.1, 'The Layout of an Activation Record'. The *runtime stack* is implemented as the class `runtime_stack` in the files runtime_stack.h[C]. Technical explanations to the stack organization can be found in the header file.

#### Instruction Execution

The *lookup table* is called `ct`, 'command table', and is implemented as an array of type `operation_descr` in the file cttable.C. Comments to the interpretation of the columns are found there. The *op-codes* are listed in the enumeration `operation_type` in the file operation.h. Finally, the *operation 'bodies'* (C++ functions) are in runtime.C.

### The Runtime Stack

The runtime stack grows and shrinks dynamically, just as needed. The interface class `runtime_stack` handles the allocation and deallocation of stack chunks (enveloped 'letter' class `st`), which are linked together as a list. To avoid memory flashing when operations happen to operate continuously across a chunk border, the last unused chunk is kept in memory.

The current stack pointer is accessed via the member function `runtime_stack::top()`. The operation bodies use mainly `runtime_stack::push()` and `runtime_stack::pop()`. For efficiency reasons, the runtime stack deals directly with the internal LWB kernel class `exprrep` rather than the programming interface for expressions, the class `expr`. This implies that the operations which manipulate the stack take care of the memory management for `exprrep` which would otherwise be provided by `expr` (see e.g. 2.1.3). The runtime stack is implemented in runtime_stack.[Ch].

### Access to Non-Local Variables

The *initialization* of the access links of local procedures is done in `proc::push_default_args()` called by `proc::init()` (both in proc.C) which in turn is called by `operation::invoke()` (runtime.C). The latter is the body of the primitive operation `invoke n`. `push_default_args()` also initializes local variables and not supplied optional arguments to the `NULLexpr`.

The access links are *used* in `runtime_stack::get_frame()` (file runtime_stack.C) which follows the required number of links. It is called from the macro `search_frame()` which is called, for instance, in `operation::push_val_from_value` (both in runtime.C), the body of the primitive `val_valarg`.

### Optional Arguments

Each LWB procedure (class `proc`, (proc.h)) is associated with a scope (class `scopeblock`, (scope.h)) which stores the minimum and the maximum number of arguments. The difference of the two numbers denotes the number of optional arguments. When the function is called, the missing arguments are initialized just as the local variables (`proc::push_default_args()` in proc.C). The number of arguments and local variables left to be initialized is the total number of room to be reserved for the parameter section of the activation record (`scopeblock::getreserve()` called by `proc::getparams()`) minus the number of effectively provided arguments.

### Code Transformation

The parsing of an lhs is described by the nonterminal `lhs_expr` in yacc.y. Note that only symbols and indexed symbols qualify as `lhs_expr`. The action code calls the corresponding special functions of the class `code_generator` (file gen.C). In gen.h the technical solution of the code transformation is described in detail.

### Interactive Mode

The parser distinguishes single statements (for execution within *input_proc*) from procedure declarations in the rule for the top nonterminal `lwbcommand` (yacc.y). Procedure definitions fall under `variacmd`.

To start the *input_proc*, a special `lwbAM::execute()` method (mach.C) pushes first a special instance of the class `proc` (proc.C) and calls then a simplified `proc::-init()` method (which is done for common procedures by the `invoke` primitive).

### Runtime Errors

The implementation of the runtime error handling can best be understood by looking at a call tree, see Fig. A.3. We look again at the example in which a linear logic formula is passed to the classical prover.

```
operation::invoke()                                   (runtime.C)
  Cfuncdescr::apply()                 descriptor keeping cpc_I_provable
    cpc_I_provable               argument check function of the prover
      invalidarg()                                (errors.h, mach.C)
        Cfuncerror()                                        (mach.C)
          lwbAM::error()                                    (mach.C)
            lwbAM::unwind()            clean up and set IP to NULL
            printerror()                                    (mach.C)
      return NULLexpr                 cpc_I_provable must return an expr
```

Figure A.3: Call tree of a runtime error case.

The class `Cfuncdescr` (see A.2.5) is an expression which basically holds a C function pointer to which the argument list is passed when the `apply` method is called. The function to be called here is the interface function to the decision procedure in the module `cpc`. Interface functions usually return a result expression, but in the error case a `NULLexpr` is returned.

The function `Cfuncerror()` sets the flag `lwbAM::erroratfuncdescr` which is checked and reset by `operation::invoke`. When returning from an error (i.e. the flag is set), the latter does *not* push the result (the `NULLexpr`).

Remark: if `lwbAM::unwind()` stops at an `catcherror()` then `lwbAM::error()` does not call `printerror()` but resets the global `error` symbol and pushes it (as the result) on the stack.

## A.3.4 The Module Concept

### The Module Description List Class `moduledescrlist`

In order to know what is to be done when a module is loaded (or unloaded), a module description of each available module must be stored somewhere. The description consists of four parts:

1. The module name (usually a mnemonic, e.g. `ipc`).
2. A short name explanation (e.g. intuitionistic propositional calculus).
3. A function pointer for the load procedure, e.g. `init_ipc()`.
4. A function pointer for the unload procedure, e.g. omitted.

The module descriptions are organized in a list (class `moduledescrlist`). In the LWB there is exactly one global instance of this class: `moddescrlist` (declared in symboltable.h). It is initialized at start-up time in the function `initmodules()` (initmodules.C). Note that both the load procedure and the unload procedure are optional.

### The Load Procedure of a Logic Module

An example of a load procedure is shown in the fragment below, where C function descriptors are installed in the module symboltable as 'protected' symbols (if the last argument is 1, the symbol is protected against assignments by the user).

```
int init_cpc( symboltable &st ) {
  ...
  st.install( "consistent" , new Cfuncdescr(cpc_I_consistent) , 1 );
  st.install( "provable" , new Cfuncdescr(cpc_I_provable) , 1 );
  st.install( "satisfiable" , new Cfuncdescr(cpc_I_satisfiable) , 1 );
  ...
  return 1;
}
```

code fragment 23, file: initmodules.C

## A.3.5 Interface Functions to Module Algorithms

The functions which are discussed in this section are used to build C function descriptors (`Cfuncdescr`, A.2.5) which are entered in a module symboltable. The C function descriptors accept only functions with a specific signature (see below). These functions form the interface to the actual module functions which may have any signature. The following naming convention for interface functions has been adopted (*mod* = module name, *algo* = module algorithm, the `I` stands for interface):

```
expr mod_I_algo(const fixedlist &args);
```

The task of an interface function is to analyze the argument list and to transform it to arguments for the actual algorithm function, which usually has the name *mod_algo*. Conversely, it has to build an expression from the *mod_algo*'s result (which need not be of type `expr`). The algorithm *mod_algo* itself should contain no argument conversions or option checking, so that is can be used efficiently by other C functions. The interface function may also serve as a dispatcher for more than one algorithm, e.g. for different variants, from which the user selects by an additional argument.

For example, the interface function to the length function for classical formulae is defined as follows:

```
expr cpc_I_length( const fixedlist &args )
{
  if ( type_one_formula(args) )
    { return int2expr(cpc_length(args[1])); }
  else
    { return NULLexpr; }
}  // cpc_I_length
```
code fragment 24, file: cpc_interface.C

The *mod_I_algo* functions are combined in *mod*_interface.[hC] (one file per module).

### Standardized Error Messages of Interface Functions

The most frequent argument error messages issued by these interface functions have been standardized (they are declared in errors.h).

1. Wrong number of arguments (`nargserror()`). The interface function reports the number of expected arguments. For example:

   ```
   cpc> pigeonhole(a,b,c);
   ERROR in pigeonhole: 1 or 2 arguments expected
   ```

2. An invalid argument was submitted (`invalidarg()`). One argument is not accepted for this algorithm, although it is syntactically correct (otherwise a parse error would have occurred, see 3.3.6). The interface function specifies the position of the invalid argument.

   ```
   cpc> pigeonhole(3, a & b);
   ERROR in pigeonhole: invalid argument 2
   ```

   To keep the message short and uniform for all cases, no reason why the argument is not accepted is given. In particular, the error message is not to explain the calling sequence(s). The detailed description of the calling sequences can be found in the help system.

# Bibliography


[1] F. Achermann. The LWB programming language. Master's thesis, IAM, University of Bern, 1995.

[2] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison Wesley, 1982.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, 1986.

[4] Owen Astrachan. METEOR: Exploring model elimination theorem proving. *Journal of Automated Reasoning*, 13(3):283–296, 1994.

[5] J. Barwise and J. Etchemendy. Visual information and valid reasoning. In *Philosophy and the Computer*, pages 160–182. Westview Press, Boulder CO, 1992.

[6] B. Beckert, S. Gerberding R. Hähnle, and W. Kernig. The tableau-based theorem prover 3TAP for multiple-valued logics. In *Proceedings, 11th International Conference on Automated Deduction (CADE), Saratoga Springs, NY*, LNCS 607. Springer, 1992.

[7] R. Bornat and B. Sufrin. Displaying sequent-calculus proofs in natural-deduction style: experience with the Jape proof calculator. Presented at PTP-97, Schloss Dagstuhl, 1997.

[8] R. Bornat and B. Sufrin. Jape: a calculator for animating proof-on-paper. In *CADE-14*, LNCS 1249, pages 412–415, 1997.

[9] S. Bose, E. Clark, D.E. Long, and S. Michaylov. Parthenon: A parallel theorem prover for non-horn clauses. *Journal of Automated Reasoning*, 8:153–181, 1992.

[10] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*, volume 23 of *Perspectives in Computing*. Academic Press, Inc., 1988.

[11] P. Brambilla. Sequent calculus proof representation. Master's thesis, IAM, University of Bern, 1996.

[12] M. Browne, E. Clarke, D. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, 35:1035–1044, December 1986.

[13] B. Char, K. Geddes, G. Gonnet, B. Leong, M. Monagan, and S Watt. *First Leaves: A Tutorial Introduction to Maple V*. Springer Verlag, 1992.

[14] J. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison Wesley, 1993.

[15] Rational Software Corporation. Unified modeling language, notation guide, version 1.0. http://www.rational.com, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, January 1997.

[16] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. In *Formal Methods in System Design*, volume 1, pages 275–288, 1992.

[17] M. d'Agostino. Are tableaux an improvement on truth-tables? *Journal of Logic, Language and Information*, 1(3):235–252, 1992.

[18] M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors. *Handbook of Tableau Methods*, chapter Tableau Methods for Modal and Temporal Logics. Kluwer, to appear. (currently available as technical report TR-ARP-15-95, Australian National University (ANU)).

[19] U. Dunker, A. Flögel, H. Kleine Büning, J. Lehmann, and T. Lettmann. Ilfa — a project in experimental logic computation. Technical Report tr-ri-94-142, Universität-Gesamthochschule-Paderborn, March 1994.

[20] R. Dyckhoff. *MacLogic. A proof assistant for first-order logic on the Apple Macintosh*, 1991.

[21] E.L.Lusk and W.W. McCune. Experiments with ROO: A parallel automated deduction system. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, LNAI 590, pages 139–162. Springer, 1992.

[22] E. Emerson. Temporal and modal logic. In J. v. Leeuwen, editor, *Handbook of Theoretical Computer Science. Volume B*, pages 995–1072. Elsevier, 1990.

[23] W. Ertel. Random competition: A simple, but efficient method for parallelizing inference systems. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, LNAI 590. Springer, 1992.

[24] A. Geist et. al. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[25] J. Almgren et al. Sicstus v3 user's manual. Swedish Institute of Computer Science, PO Box 1263, S-164 28 Kista, Sweden, 1995.

[26] M. Carlsson et al. Sicstus prolog user's manual. Technical Report T91:11B, Swedish Institute of Computer Science, 1991.

[27] R. Constable et. al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[28] M.J. Fischer and R.L. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18:194–211, 1979.

[29] M. Fitting. *Proof Methods for Modal and Intuitionistic Logics.* Reidel, Dordrecht, 1983.

[30] H. Fujita and R. Hasegawa. A model generation theorem prover in KL1 using a ramified-stack algorithm. Technical Report TR-606, ICOT, Tokyo, Japan, 1990.

[31] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *Protocol Specification Testing and Verification*, volume XV, pages 3–18. Chapman & Hall, 1996.

[32] F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for ALC. In L. Aiello, J. Doyle, and S. Shapiro, editors, *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, pages 304–314, Cambridge, MA, 1996.

[33] P. Godefroid, G. Holzmann, and D. Pirottin. State space caching revisited. In *Workshop on Computer Aided Verification*, Montreal, 1992.

[34] A. Goerdt and U. Kamps. On the reasons for average superlinear speedup in parallel backtrack search. In *Proceedings CSL 93*, LNCS 832, pages 106–127. Springer, 1994.

[35] R. Goré. *Cut-free Sequent and Tableau Systems for Propositional Normal Modal Logic.* PhD thesis, Computer Laboratory, University of Cambridge, England, 1992.

[36] G. Gough. Decision procedures for temporal logic. Technical Report UMCS-89-10-1, Department of Computer Science, University of Manchester, 1989.

[37] R. Griswold and M. Griswold. *The Implementation of the Icon Programming Language.* Princeton University Press, 1986.

[38] R. Hähnle. *Automated Deduction in Multiple-valued Logics.* Oxford Science Publications, 1993.

[39] André Heck. *Introduction to Maple.* Springer Verlag, 1993.

[40] D. Heller and P.M. Ferguson. *Motif Programming Manual*, volume 6 A of *The Definite Guides to the X Window System.* O'Reilly & Associates, Inc., 1994.

[41] A. Heuerding and S. Schwendimann. A benchmark method for the propositional modal logics K, KT, S4. Technical Report IAM-96-015, IAM, University of Bern, 1996.

[42] 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.

[43] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq proof assistant, a tutorial, version 5.10. Technical Report RT-0178, INRIA, Rocquencourt, France, 1995.

[44] U. Hustadt and R. Schmidt. On evaluating decision procedures for modal logic. Technical Report MPI-I-97-2-003, Max Plank Institut für Informatik, 1997.

[45] C. Jard and T. Jeron. Bounded-memory algorithms for verification on the fly. In *Workshop on Computer Aided Verification*, Aalborg, 1991.

[46] B. Kernighan and R. Pike. *The Unix Programming Environment.* Prentice-Hall, 1984.

[47] Y. Kesten, Z. Manna, H. McGuire, and A.Pnueli. A decision algorithm for full propositional temporal logic. In *Computer Aided Verification*, LNCS 697, pages 4–35. Springer, 1993.

[48] R. Letz, S. Bayerl, J. Schumann, and W. Bibel. SETHEO - a high-performance theorem prover. *Journal of Automated Reasoning*, 8:183–212, 1992.

[49] J. Levine, T. Mason, and D. Brown. *lex & yacc.* O'Reilly & Associates, Inc., 1992.

[50] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *12th Ann. ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.

[51] S. Lippman. *C++ Primer.* Addison Wesley, 1993.

[52] Z. Luo and R. Pollack. *LEGO Proof Development System: User's Manual.* Department of Computer Science. University of Edinburgh, 1992.

[53] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1991.

[54] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[55] A. Nye and T. O'Reilly. *X Toolkit Intrinsics Programming Manual*, volume 4 of *The Definite Guides to the X Window System.* O'Reilly & Associates, Inc., 1993.

[56] F. Oppacher and E. Suen. Harp: A tableau-based theorem prover. *Journal of Automated Reasoning*, 4:69–100, 1988.

[57] L. Paulson. *Isabelle: A Generic Theorem Prover.* Number 828 in LNCS. Springer, 1994.

[58] A. Heuerding R. Goré, W. Heinle. Relations between propositional normal modal logics: an overview. *Journal of Logic and Computation*, 7(5):649–658, 1997.

[59] D. Sahlin, T. Franzén, and S. Haridi. An intuitionistic predicate logic theorem prover. *Journal of Logic and Computation*, 2(5):619–656, 1992.

[60] J. Schumann. Parallel theorem provers - an overview. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, LNAI 590, pages 26–50, 1992.

[61] A. Sistla and E. Clarke. The complexity of propositional linear temporal logic. *Journal of the Association for Computing Machinery*, 32(3):733–749, 1985.

[62] C. Suttner. A parallel theorem prover with heuristic work distribution. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, LNAI 590. Springer, 1992.

[63] C. Suttner. *Parallelization of Search-based Systems by Static Partitioning with Slackness.* PhD thesis, Institut für Informatik, TU München, February 1995. Published by Infix-Verlag, volume DISKI 101.

[64] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[65] S. Wolfram. *The Mathematica Book, Third Edition.* Wolfram Media, Inc. and Cambridge University Press, 1996.

[66] P. Wolper. The tableau method for temporal logic: an overview. *Logique et Analyse*, 110-111:119–136, 1985.

[67] L. Wos. *The Automation of Reasoning: An Experimenter's Notebook with Otter Tutorial.* Academic Press, 1996.

# Index

# Lebenslauf

| | |
|---|---|
| **1966** | Geboren am 20. 12. in Thun (Be). |
| **1973 — 1982** | Primar- und Sekundarschule Belp. |
| **1982 — 1986** | Städtisches Literargymnasium Bern-Kirchenfeld (Matur Typus A). |
| **1986 — 1993** | Informatik-Studium an der Universität Bern:<br>Lizentiat am 14. Jan. 93, Titel der Arbeit: 'Ein Softwarepaket für die Algebraische, Projektive Geometrie in Maple'.<br>1. Nebenfach: Mathematik, 2. Nebenfach: Geographie. |
| **1993 — 1995** | Wissenschaftlicher Mitarbeiter an der Universität Bern:<br>Nationalfonds-Projekt SPP IF: 'A Logics Workbench' (Nr. 5003-34279). |
| **1996 — 1998** | Doktorand/Assistent an der Universität Bern bei Prof. G. Jäger:<br>Nationalfonds-Projekt 'Distributed Propositional Proof Systems' (Nr. 21-43197.95). |