Integrating Logical Functions with ILF

Bernd I. Dahn
Jürgen Gehne
Thomas Honigmann
Lutz Walther
Andreas Wolf

Institut für Reine Mathematik der
Humboldt-Universität zu Berlin
Ziegelstraße 13a
10099 Berlin
Germany

March 1, 1994

Abstract

This is a description of the system *ILF* developed at the Humboldt-University at Berlin[1]. *ILF* is a system that integrates automated theorem provers, proof tactics for interactive deductive systems and models within a graphical user interface. The structure and commands of *ILF* are presented[2].

A special part is devoted to the *TreeViewer* – a part of *ILF* used for visualising directed acyclic graphs, which can be used as a separate programme.

We describe the possibilities to extend *ILF* by integrating more interactive and automated deductive systems.

The last part describes the *ProofPad* – a sample configuration for editing proofs in the field of lattice ordered groups.

[1]This work was supported by the Deutsche Forschungsgemeinschaft within the project "Deduktion und verbandsgeordnete Gruppen"

[2]for further information contact `e-mail: gehne@mathematik.hu-berlin.de`

Contents

Chapter 1

What is ILF?

Research in the field of theorem proving in many groups in several countries has created a lot of sophisticated tools e.g.

- automated theorem provers for various logical calculi,
- rewrite systems,
- proof tactics,
- model finders and
- domain specific methods.

ILF is a tool that can be configured in many ways to *I*ntegrate all these *L*ogical *F*unctions. The common feature of these tools that is used for this integration is that they all can be used to modify a knowledge base.

ILF is applied on two different levels. For the knowledge engineer it yields methods of testing the power of tools to support logically correct arguments in a specific field. Several ways to combine these tools in proof tactics can be tested rapidly. When a collection of useful proof tactics has been obtained, it can be encapsulated as a set of "rules of inference" in a new interactive or automated deductive system. It is also possible to extend an existing system in this way. This new, more powerful system can be tailored to meet exactly the needs of an end user, making available just those procedures that his kind of problems demand.

The user of *ILF* is not restricted to a particular logic. He can use any of the systems that have been integrated at any time. He might start proving a theorem within one calculus. On the way he can decide to prove a specific formula by a specialized system. E.g. he may apply a rewrite system in order to prove an equation and apply this equation to construct a natural deduction proof.

The user can take advantage of the programming language PROLOG to write proof tactics for interactive deductive systems. These tactics can change the state of a proof by applying rules of inference in an automated way. PROLOG is augmented by special constructs – socalled tacticals, loop constructs and global variables – in order to help the user writing his own control programme for a theorem prover. His proof tactics can ask advanced automated theorem provers running in the background for support. Different alternatives for a proof can be tried automatically using the backtracking capabilities of PROLOG. All the debugging tools of PROLOG like tracing and spy points are available to develop and test proof tactics.

Perhaps the most challenging feature of *ILF* is its modularity. The power of *ILF* can be easily extended by integrating further systems and developing libraries of domain specific proof tactics. In fact, for an experienced PROLOG programmer, it is a matter of a few days to integrate a new system that has been developed somewhere else independently.

The continuously growing power of *ILF* requires a simple tool for control. This is provided by a graphical user interface. The functionality of this interface can be changed and modified by the user at run time; allowing also automatic modification by the user's proof tactics.

A common problem is to prevent the user from getting lost in complex hierarchies of objects, e.g. in a proof, a complex term or a hierarchy of theories. Therefore, *ILF* contains a *TreeViewer* that can be used to visualize these hierarchies as marked directed acylic graphs. Though this *TreeViewer* was developed as a part of *ILF*, it is in fact a separate programme with a simple interface to be used in combination with other programmes.

The development of *ILF* at the Humboldt-University started around 1987 as an interactive theorem prover with a language for proof tactics. In 1989 experiments with the combination of several deductive systems began. In 1991 *OTTER* from Argonne National Laboratories became the first automated theorem prover to be integrated into *ILF*. 1990-1992 the development of *ILF* was supported by the Volkswagen-Stiftung within a joint project "Leistungsfähigkeit von Beweisstrategien" with the group of H. Kleine Büning from Duisburg University. IBM Germany made *ILFA* available – a toolkit for developing deductive systems in C. Using this system, interactive provers for the modified problem reduction and for the tableau calculus were built for *ILF*. Since 1992 a project "Deduktion und verbandsgeordnete Gruppen" supported by the Deutsche Forschungsgemeinschaft has provided the framework and goals for the development of the system. This project is part of the DFG-Schwerpunkt "Deduktion". Cooperation within the Schwerpunkt made the provers *DISCOUNT* (University of Kaiserslautern) and *SETHEO* (Technical University Munich) available to *ILF*. We are greatly indebted to the authors of these systems for their assistance in integrating these provers into *ILF*.

Also the development of the *TreeViewer* has been stimulated very much by discussions at workshops on proof tactics within the Schwerpunkt.

Subsequently, the structure and the commands of *ILF*, as they were implemented in February 1994, are described.

A sample configuration – the *ProofPad* – is explained. This "deductive system" is set up to assist a user without special knowledge in automated theorem proving in editing elementary proofs in the field of lattice ordered groups, making the best possible use of the power of automated theorem provers.

For the sake of extending the power of *ILF*, the interfaces needed for integrating other deductive systems are discussed.

We do not presume that the reader has experience with *ILF*, but some knowledge on PROLOG, on the deductive systems and on the logical calculi used by *ILF* will be helpful. The corresponding references are either generally available or can be easily obtained. Therefore, we shall only sketch these aspects.

Chapter 2

Installation and Configuration

ILF is available from the *ILF* group as a compressed tar-file `ilf.tar.Z`. By uncompressing and x-taring `ilf.tar.Z`, the directory ilf and further subdirectories (e.g. the `bin` , `man` and `tmp` directories) will be created.

ILF is based on *PROLOG-2* from Expert Systems Ltd, so *PROLOG-2* must be available. It is recommended to install also the automated provers *SETHEO* [LSBB92] and *DISCOUNT* [DP92] and use them as described in Section 8.2.2 and 8.2.1.

By convention all executables in *ILF* except for *ilf* have the extension `.exe` in order to prevent PROLOG-2 from automatically adding the extension `.pro`.

The following environment variables must be set:

environment variables

`ILFHOME`	the home directory of *ILF*, usually the directory built by x-taring the `ilf.tar` file
`USERILFHOME`	the personal *ILF* directory owned by the user, in which *ILF* will store temporary files and configurations and modifications for a particular user
`PROLOGHOME`	the home directory of PROLOG-2
`TVIEWHOME`	the home directory of the treeviewer

The executable files are in `$ILFHOME/bin` and the manpages are in `$ILFHOME/man`. `$ILFHOME/bin` and `$ILFHOME/man` should be included in the user's `PATH` and `MANPATH` variable, respectively.

The entries of the manpages normally start with `ilf_` and are located in the `l`-section of the manpages. The complete list of *ILF*-manpages can be found in the `ilf_man` manpages.

Since temporary files will be created during an *ILF* session, a user should have a personal *ILF*-directory `$USERILFHOME` different from the *ILF* directory `$ILFHOME` in order to avoid conflicts with filenames. The current version requires a special structure of the `$USERILFHOME` directory, which can be created by running `ilf` with the `-ini` option.

ILF is envoked by `ilf [option [option ...]]`, where the currently available options are the following:

ILF command line options

`-h`	show the possible options
`-hosts`	create a new list of actual hosts
`-ilfhome IH`	take `IH` as `ILFHOME`
`-ilfrc IRC`	take `IRC` as `.ilfrc` file
`-ini`	create the `USERILFHOME` directory using the environment variable `$USERILFHOME` and all necessary subdirectories
`-prologhome PH`	take `PH` as `PROLOGHOME`
`-t`	test the environment, don't start *ILF*
`-userilfhome UIH`	take `UIH` as `USERILFHOME`
`-x`	run *ILF* in an X-Window system
`+x`	do not run *ILF* in an X-Window system
`-xilf`	take the executable `XILF` instead of `Xilf`, the default executable

For the actual list of options see the `ilf_ilf` manpages or the `README` file. Further options (e.g. X-toolkit options) will be passed to `Xilf`.

ILF will first read the file `.ilfrc` in `$USERILFHOME`. If this file does not exist, *ILF* will take the generic `.ilfrc` in `$ILFHOME`.

The entries of the `.ilfrc` file are divided into groups, each group `GROUP` is enclosed in the keywords `GROUP` and `END_GROUP`. There are at least the groups `ILFSTATE`, `EXPERT`, `DEDSYS` and `SIGNATURE` with the following syntax:

ILF_STATE
key$_1$ value$_1$
key$_2$ value$_2$
. . .
key$_n$ value$_n$
END_ILF_STATE

EXPERT
expert$_1$ [file$_1$]
expert$_2$ [file$_2$]
. . .
expert$_n$ [file$_n$]
END_EXPERT

```
DED_SYS
ded_sys1 [file1]
ded_sys2 [file2]
.  .  .
ded_sysn [filen]
END_DED_SYS

SIGNATURE
op1 ass1 prec1
op2 ass2 prec2
.  .  .
opn assn precn
END_SIGNATURE

TEX_OP
op1 ass1 prec1 texstring1
op2 ass2 prec2 texstring2
.  .  .
opn assn precn texstringn
END_TEX_OP
```

The entries of the groups will be stored as PROLOG predicates:

- the `ILF_STATE` entries as `ilf_state(key`$_i$`,value`$_i$`).`
- the `EXPERT` entries as `expert_file(expert`$_i$`,file`$_i$`).`
- the `DED_SYS` entries as `dedsys_file(dedsys`$_i$`,file`$_i$`).`
- the `TEX_OP` entries as `tex_op(op`$_i$`,ass`$_i$`,prec`$_i$`,texstring`$_i$`).`

If no file is given, `none` is taken as the file. These files are consulted when the corresponding experts and deduction systems, respectively, are called. The entries in the `TEX_OP` group define the LaTeX representation of these operators.

```
KILL
prog1
prog2
.  .  .
progn
END_KILL
```

forces *ILF* to kill all processes, where `grep` finds the pattern `prog`$_1$,...,prog$_n$ in the output lines of `ps -auxww` after it has finished.

For distributed computation, the list of possibly available hosts is to be held in the file `.ilfhosts` in the `USERILFHOME` or in the `ILFHOME` directory. It is assumed that the `ILFHOME` and `USERILFHOME` directories are mounted at these hosts and that the user can login without password checking (i.e. the hosts should be in the users `.rhosts` file). *ILF* checks which of these hosts are alive and stores the names of these hosts for further use in the file `USERILFHOME/tmp/ilfhosts.pro`. As long as this file exists, *ILF* assumes it to be up to date, so further invocation of *ILF* will skip the check. In order to update the list of actually available hosts, the user should simply remove the file `USERILFHOME/tmp/ilfhosts.pro` or just start *ILF* with the `-hosts` option. It is not recommended to change any file in `USERILFHOME/tmp` during an *ILF*-session.

ILF will start two processes running PROLOG-2, one in the foreground to react to the user's command and one in the background to control the work of automated systems running as separate processes. Depending on the automated systems used, more processes may be created. All files automatically created by *ILF* at runtime are located in the directory `$USERILFHOME/tmp`. Files specific for a particular deductive system `SYSTEM` are kept in `$USERILFHOME/dedsys/SYSTEM`. Theories and tactics are contained in `$USERILFHOME/th` and `$USERILFHOME/tac`. They have the extensions `.th` and `.tac`, respectively.

The behaviour of *ILF* depends very much on the setting of *ilf_states*, which are explained at appropriate positions in this paper. Any *ilf_state* used during a session must be initialized in the file `.ilfrc` described above. Then it can be retrieved by the command `ilf_state(key,Value)` and set by the command `ilf_state(key,OldValue,NewValue)`.

The declaration of operators in the `SIGNATURE` section of the `.ilfrc` file follows the conventions of PROLOG-2. Note that changing these settings may change the way *ILF* parses theories and tactics! When states of background systems that have been saved before using the appropriate commands are reloaded, *ILF* checks whether the relevant operator declarations have been changed and displays a warning message if a change has been detected.

Chapter 3

The User Interface

Unlike other deductive systems *ILF* is normally used with a graphical user interface. The main reason for this is that the integration of many systems confronts the user with a lot of different commands that can be applied. The graphical user interface is an efficient help for the user in order to find out which commands are available in a specific situation. Since these commands – when selected from a menu – are printed in the command line, the graphical user interface can assist in learning the commands necessary to write a proof tactic. Nevertheless, there is also a command line interface available. Both interfaces are described below.

3.1 The Command Line Interface

The command line interface of *ILF* is essentially the top level interpreter of the underlying PROLOG system. Thus any PROLOG-2 command can be run from the command line interface.

The graphical user interface is just a shell on top of the command line interface; any command given to the graphical user interface will be delivered to the command line interface. However, it is also possible to work only with the command line interface. In order to do this, the *ilf_state* `x` must be set to `off` – either by including an appropriate statement in the `.ilfrc` file or by starting *ILF* with the command line option `+x`.

Another *ilf_state* the user may want to set is `exptty`. If this *ilf_state* is set to `file`, the whole output from the background expert communicator is then sent to `file`. The default value for `exptty` is `/dev/tty`, i. e. the terminal the user is working on. Since the background experts work asynchronously, the default behaviour may be confusing when working with the command line interface. The *TreeViewer* described in Section 4 is an X11–based application that is not available from the command line interface.

There are very few facilities that are only available at the command line interface due to the way Motif handles the user input. First of all, there is no way to interrupt a running prolog programme from the graphical user interface. From the command line interface this can be done by typing the terminal's INTR character (normally `CTRL-C`). From the command line interface the EOF character (`CTRL-D` if not specified otherwise) can be used to signal the end of a file (or a break for that matter). When using the graphical user interface `end_of_file` has to be typed or the corresponding menu item must be chosen. When tracing through a programme, going to the next step on the command line interface is performed by typing `RETURN`, where the graphical user interface requires `SPACE` and `RETURN`.

Subsequently we describe the most important command line interface commands, which are not specific for particular deductive systems. They are given in the full predicate notation `predicate/arity`.

`load/1` loads theories from the `th`–subdirectory of `$USERILFHOME` and tactics from the `tac`–subdirectory.Whether to load a theory or a tactic is decided by the extension (`.th` vs. `.tac`) of the given filename.

`load_th/1` and `load_tac/1` load a theory or a tactic located elsewhere if called with an absolute path as argument.

`load_th/0` and `load_tac/0` display a list of all theories/tactics available in the user's `th/tac`–subdirectories and prompt the user with `Choice:` to enter a number.

After loading a theory, `load_th/0` and `load_th/1` allow the user to enter additional axioms. A number prompts for the axiom in a form described in Section 5, `Name` prompts for the name of the axiom. An `end.` typed to the number prompt stops the input.

`forget_th/0` clears the knowledge base. `axioms/0` displays all axioms.

The commands

```
Head to Body by Tactic
Name reduces Head to Body by Tactic
```

are central to *ILF*. `Tactic` is executed to prove that `Body` implies `Head`. The call succeeds if `Tactic` succeeds and proves an instantiation of `Body` → `Head`. In general `Head` and `Body` may contain variables for subformulas that are instantiated during the proof if this is supported by the particular deductive system.

`reduces` asserts the proved instantiation under the name `Name` into the knowledge base.

The most simple case of a tactic is to call a deductive system by its name as a 0-ary predicate. This means that the user will interactively construct a proof by means of the specified deductive system.

Sometimes the lack of the occur check in PROLOG can cause problems when unifying formulas or names of axioms. These problems can often be overcome by using the command `occur/0`, which switches the occur check for some operations on or off. The actual status of this flag is contained in the *ilf_state* `occur`.

The `halt/0`–command leaves the PROLOG system and therefore the command line interface of *ILF*, too.

3.2 The GUI Windows, Menus and Commands

The *ILF* System can be used within a graphical interface that is based on the X11R5 and OSF/Motif 1.2 libraries. Using the graphical user interface, the system consists of the following windows. Some of these windows can be omitted by setting appropriate *ilf_state*s described below. Notably, the graphical user interface can be switched off by setting the *ilf_state* `x` to `off`.

- The **main window** controlling the PROLOG-2 system running in the foreground. It has pulldown menus, an area for the *stdout* stream of PROLOG and a line for editing.
- The **ExpertManager window**, which displays messages from the PROLOG system running in the background and from the automated systems called by that PROLOG. The window is an ordinary xterm. The ExpertManager and its window can be closed by typing `ex_stop.` in the main window. The manager can be restarted (a new PROLOG process will be created) by `ex_start.`
- The **expert log window**, which is used to display some information for debugging. This window is only displayed if the *ilf_state* `debug` is set to `on` in the `.ilfrc` file. While *ILF* is running, this *ilf_state* switches the displaying of messages for debugging on or off. If the window is not present, and the state is set `on`, the debug messages are written into the file `exlog.l` in the `tmp` directory.
- The two **windows of the TreeViewer** that are described in Section 4.

In the graphical user interface, too, all commands can be given to the system by typing them on the command line of the main window, but it is easier to use the menus. There is a standard set of commands given in the menu bar and the pull down menus there. Which menu items are displayed depends on the *ilf_state* `menu_level`. The level 0 will give only the basic menu items, and a higher level will give more menu points. At present 4 is the maximum menu level in use. Menus with a variable number of items contain an item `refresh`, which can be used to adapt the menu to a new menu level. It is also possible to introduce new menus (see Section 9).

If a command is selected from a pulldown menu, its command line form is shown in the main window. Some commands must be completed by the user by inserting certain parameters. In these cases the displayed format of the commands indicates the types of the parameters required.

After the *ILF* system has been started, it usually has four menus: the `COMMAND` menu, the `THEORY` menu, the `DED-SYS` menu, and the `expert` menu. The `COMMAND` menu has some items that can be used in several systems to edit a proof. These commands are explained in Section 6. Moreover, there are the following general commands.

COMMAND menu

`occur-check`	n[1]	switches the occur-check for some unifications in the foreground PROLOG on or off
`load tactic`	c	opens a file select box of the user's tactics
`halt`	n	leaves the *ILF* system and kills all subprogrammes

The `THEORY` menu permits access to commands to inspect and manipulate the hierarchy of theories in use. These commands are discussed in Section 5.

The interactive deductive systems, which will be decribed in Section 8.1, can be called from the menu `DED-SYS`.

[1] n: noconfirm, c: confirm, s: submenu

Chapter 4

The TreeViewer

The *TreeViewer* is a graphical and alphanumerical interface for the visualisation of directed acyclic graphs. It is a separate programme that can be integrated into other programmes. The *TreeViewer* can be obtained via ftp from `info.mathematik.hu-berlin.de` (141.20.54.8, not yet connected to a name service) in the compressed tar-file `/ftp/ilf/tview.tar.Z`.

It is based on the OSF/Motif library and should therefore work on any X-Window-system on SUN SPARCStation and compatibles. In *ILF* the *TreeViewer* is used if the *ilf_state* `treeviewer` is set to `on` in the file `.ilfrc`.

The graphs to be displayed are given by handles for the nodes, especially a designated root handle, and edges. Each handle for a node can be optionally associated with a shape and an info displayed within the graph and a contents string, which can be displayed in a separate window.

Having been invoked by the user, the *TreeViewer* offers two windows: one for the graphic display of the graph and one for alphanumeric output and editing.

In the graphic window, by default graphs are displayed unfolded as trees showing nodes occurring more than once in several places. Such nodes are marked by a bold line in the drawn rectangle. The user can gather these nodes by commands chosen from the pulldown menus of the graphic window. Of course, they can also be separated again by an appropriate command. According to the shape associated with a handle, the node is displayed as one of 10 bitmaps. These bitmaps are contained in the file `tvshapes`, which can be edited by the user with a bitmap editor, e.g. `bitmap`.

It is possible to hide parts of the graph and to redraw hidden parts.

There are several possibilities to display alphanumerical information about the nodes. First of all, there is an info shown on the right–hand side of the nodal rectangle. It is set automatically, but can be modified by the user (see commands). A further way is to click on a node in the graphic window using the left mouse button. This method will show the handle of the node in the bottom line of the graphic window.

In the alphanumeric window, textual information is displayed in the form `info-string:` spaces `handle` or `contents` if set, respectively. The number of spaces depends on the depth of the node in the graph, thus indicating the structure of the graph in the alphanumeric window, too. The information about a node is transferred to the list in this window by clicking on the node in the graphic window with the middle mouse button. Clicking again will remove the information from the list. If information on the node is displayed in the alphanumeric window, the rectangle visualising this node has a shadow.

Another way to get a line into the above mentioned list is to use the `tv_text` command (see commands). In this case the string `"user"` is shown instead of the info-string and the number of spaces is constant. Handle or contents, respectively, can be edited for further use in the bottom line of the alphanumeric window. They are displayed in the bottom line after selection with the left mouse button.

The communication of the *TreeViewer* with the programme having invoked it is realized by the use of two named pipes, one for the input, one for the output. These pipes can be created by the client programme before invoking the *TreeViewer* or initiated by the *TreeViewer* if it is called with the `c` option. In this case they are removed automatically when the *TreeViewer* is closed.

The *TreeViewer* is called by

```
tview [-l] [-c] inputpipename outputpipename ...
```

The option `-l` makes the programme write a logfile into the working directory, which contains all commands received through the inputpipe and errors generated by these commands if occurring.

The option `-c` causes the programme to generate communication pipes and remove them when being closed. `Inputpipename` and `outputpipename` are the names of the pipes for communication with the client programme. At the end of the option list further options as generally known from graphical interfaces like `-geometry` or `-iconic` can be added.

The client programme communicates with the *TreeViewer* using the following *commands*. All these commands are given to the *TreeViewer* through the inputpipe.

TreeViewer commands for building the graph

`root(Root)`	creates a new root-node with the handle `Root` removing an existing graph if nesseccary
`new_edge(Father,Son)`	creates a new edge in an existing graph if `Father` is known, the edge doesnt yet exist and `Son` is not already located on the direct way from `Father` to `Root` (i. e. the graph must be acyclic)
`del_edge(Father,Son)`	deletes an edge of the graph if it exists and `Son` is a leaf (i. e. has no sons)
`set_shape(Handle,Shape)`	sets a shape for the node `Handle`, the number of which must be in the range from 0 to 9
`set_info(Handle,Info)`	sets the `info` for the node `Handle`

`set_contents(Handle,Cont)` sets the contents `Cont` for the node `Handle`

`Handle`, `Contents` and `Info` are treated as strings, in case they contain a comma (which is used as separator between the arguments of commands) they must be enclosed in double quotes. Quotes are taken as a part of the string if they are preceeded by a backslash.

The *communication* of the *TreeViewer* with the client programme can be performed by the following commands. All messages from the *TreeViewer* are finished with a full-stop.

TreeViewer commands for communication

`ok(message)`	will cause the treeviewer to respond with `message.` using the outputpipe
`quit`	closes the *TreeViewer*
`get_handle`	has the *TreeViewer* write the next node handle and contents, respectively, selected by the user to the outputpipe. The node has to be selected in the graphic window with the right mouse button. This is signalled to the user by the form of the mouse-cursor appearing as a hand.
`get_text`	initiates an answer containing facts in the form of a list of PROLOG terms – each item being on a separate line – surrounded by the keywords `tv_list.` and `end.`. The user may select a string in the editable line of the alphanumeric window end has to press the `Return` key. Then the following information is passed to the client programme: `handle` of the node corresponding to the selected line in the text, `position` of the first character of the selected substring (the first symbol has the position 0), `length` of the selected substring and the `string` in the editable line surrounded by quotes and followed by a full-stop. If there is not any part of the string selected, the value of position is set to -1 and the length to 0.
`get_info`	works as if both `get_handle` and `get_text` have been called, but only one answer is allowed. The user decides how to give the answer.
`tv_text(message)`	generates a line containing `"user:  message"` in the alphanumeric window
`update`	refreshes the contents of both the graphic and the alphanumeric window after having changed things by commands using the inputpipe

`/* ... */`	is a comment to be ignored
`call`	allows the client programme to call functions usually invoked from the menus by using the inputpipe

TreeViewer menu in the graphics window

`Hide`	to remove nodes from the drawing, sons possible being displayed as sons of the father node
`Above`	hides all nodes above the selected one (excepting the root-node)
`This`	hides the selected node (root-node cannot be hidden)
`Below`	hides all nodes below the selected one
`Below but Leafs`	hides all nodes below the selected one except for the leaves (the nodes without sons)
`Draw`	displays nodes that have been hidden
`Above`	displays hidden nodes above the selected one
`Below`	displays hidden nodes below the selected one
`Gather`	displays nodes occurring several times in the unfolded graph only once
`This`	contracts only the selected node
`All`	contracts the selected node and all nodes having more than one instance below it
`Separate`	unfolds the graph after having used gather
`This`	displays the selected node as often as it occurs as a son
`All`	displays the selected node and all nodes below it as often as they occur as sons
`List`	transfers textual information to the alphanumeric window
`This`	transfers information on the selected node (like using the middle mouse button)
`All`	transfers information on the selected node and all nodes below it

TreeViewer menu in the alphanumeric window

`Discard`	Using this menu abandons the list in the alphanumeric window.
`Action`	erases the list
`Abandon`	cancels the action and leaves the list intact

Instead of using the menu buttons, the client programme can send the command

`call(ButtonFunction,Handle)`

through the inputpipe. The first argument `ButtonFunction` is a concatenation of the labels of the buttons that would have to be clicked with the mouse if they were used instead (e.g. the ButtonFunction for Hide-Above is `HideAbove`). The use of capitals is irrelevant so that e.g. `hideAbove` may also be used instead of `HideAbove`. The second argument `Handle` is the handle string. The argument must exist, if no handle is needed, it is not used. The difference of effects achieved by using the mouse and the buttons and the command `call`, respectively, is caused by the impossibility of distinguishing between several instances of one node without the mousepointer. That's why the command `Hide-This` for example can hide one instance of a node only if the mouse is used, in case the command `call` is used, all instances of the node are `hidden`. On the other hand, this difference allows to realize the function `Draw This` for the command `call`, which would be impossible to use with the mouse.

In order to run the *TreeViewer* the environment should be set up as follows.

For the graphic display the bitmap-file `tvshapes` must be available. This file is looked for in the actual directory and then in the directory given by the environmental variable `TVIEWHOME`.

For trouble-shooting the `l` option should be used. This causes the generation of the file `tview.log` containing all commands sent through the inputpipe. Commands not accepted are followed by a comment giving the reason of the missing acception. The file can be used as input by sending it through the inputpipe after deleting the last line containing the command `quit`.

Chapter 5

Theories

This section describes the syntax of theory files for *ILF*. A theory file may contain named axioms, a title, comments and PROLOG commands. It must contain an end line containing `end.` as the only item.

An axiom is given on two lines. The first of these lines contains a term of the form `H:-B.`, `H.` or `:-B.`. `H` and `B` are valid PROLOG terms, possibly with variables. The intended meaning of this axioms is the universal closure of $\mathtt{B} \rightarrow \mathtt{H}$, of `H` and of $\neg\mathtt{B}$, respectively. Like in PROLOG, conjunction and disjunction ar denoted by a comma and a semicolon, respectively. Variants of conjunction and disjunction, which can take any list of propositions as arguments, are denoted by `&` and `#`. Negation is denoted by `not`. Universal and existential quantifier are written as `ex` and `all`. They can bind either a single variable or a list of variables. As in PROLOG variables start either with a capital letter or with an underscore character. It is possible to use operators in prefix, postfix or infix notation if they have been declared in the file `.ilfrc`.

Of course, it depends on the properties of the concrete deductive systems which axioms can be used in a proof.

Each axiom is followed by a line containing the name of the axiom which can be an arbitrary prolog term. Clever naming conventions can simplify writing tactics in *ILF* considerably, so names should be chosen with care. E. g. all laws of commutativity can be denoted by names of the form `comm(Op)` where `Op` is the operator which is commutative. Similarly `dis(*,+,1)` and `dis(*,+,2)` can be names for the equations saying that `*` distributes over `+` from left or right, respectively.

The (optional) title of a theory is given on a single line as

`th_title :  title_string.`

`title_string` must be enclosed in double quotes. It is recommended to chose `title_string` in such a way that "Theory of `title_string`" is meaningful.

Comments are given as in C–programmes. A line containing a PROLOG command starts with `?-`. An end line is a line consisting only of `end.` and must be in any theory file.

After this informal description we give a Backus–Naur–notation for this syntax. Terminal symbols are `prolog_term`, which stands for a valid PROLOG term, `string`, which stands for any string of characters, comment_string, which stands for any string of characters not including */, <lf>, which stands for a line feed and :- ?- : . /* */ " th_title end, which stand for themselves.

ilf_theory	::=	{theory part}* end_line
theory_part	::=	empty_part ‖ axiom_part ‖ title_line ‖ comment
empty_part	::=	{<lf>}*
axiom_part	::=	axiom.<lf>prolog_term.<lf>
axiom	::=	prolog_term :- prolog_term ‖ prolog_term ‖ :- prolog_term
title_line	::=	th_title : "string".<lf>
comment	::=	/* comment_string */
end_line	::=	end.<lf>

As mentioned above, theories can contain arbitrary PROLOG commands. The command load("Name.th"). which loads a theory Name from the user's directory of theories is of special significance for theories. This theory is considered as a subtheory of the given theory. In this way, a simple hierarchy of theories is built up. Though a theory can be a subtheory of several other theories it will be loaded only once into memory. If there is a theory named standard.th in the directory ${USERILFHOME}/th, it will be loaded at startup.

Within *ILF*, theory/1 and subtheory/2 can be used to inspect the hierarchy of theories. in_theory(Formula,Theory) tests whether a formula has been loaded with a particular theory. The following commands, which can be also accessed using the THEORY menu of the graphic user interface, can also manipulate theories.

THEORY menu

make_theory(Theory)	creates a new theory
make_subtheory(Th,SubTh)	determines that SubTh is considered to be contained in Th
make_title(Th,String)	lets String be the title of Th
add_ax(Theory,AxList)	adds the axiom or the list of axioms AxList to the theory Theory
rm_ax(Theory,AxList)	removes the axioms from AxList from Theory but leaves them in the knowledge base
rm_theory(Theory)	removes Theory leaving its axioms still in the knowledge base
add_th	gives the user the possibility to add new axioms from the command line without inserting them into a particular theory
activate(Name)	activates axioms whose name matches Name. Deactivated names will not be used by deductive systems. When an axiom is deactivated, its name is changed from Name to $(Name)
deactivate(Name)	deactivates axioms whose name matches Name

`forget_ax(Name)`	removes all axioms having a name matching `Name` from the knowledge base
`forget_th`	eliminates all axioms and theories from memory
`save_th(Theory)`	is used to save a theory to the user's directory of theories
`save_th_as(Th,NewTh)`	is used to save a theory to the user's directory of theories under the name `NewTh`
`save_changed_ths`	asks the user for each theory that has been changed whether it should be saved or not
`axioms`	displays all axioms in memory on the terminal or in the command window

The *ilf_state* `save_th` can contain a list of theories that are to be saved automatically when *ILF* is left. From the graphic user interface additional tools for the presentation of theories are available.

more of the THEORY menu

`view_th`	displays the hierarchy of theories in the windows of the *TreeViewer*. If a particular theory is selected with the right mouse button, it is displayed by the LaTeX system. Selecting the top node quits this operation
`view_theory(Theory)`	displays a particular theory by the LaTeX system

Chapter 6

Proof Tactics

Proof tactics are a way to automate parts of the work to be done to build a proof in a deductive system. They can perform a fixed sequence of steps like a macro or a refined control strategy like an automated theorem prover. In general, tactics are developed to test different procedures to control the search for a proof. Approved tactics are frequently incorporated as new rules of inference into a deductive system. Tactics are kept in files in the directory `${USERILFHOME}/tac` having the extension `.tac`. If there is a file named `standard.tac`, it is reconsulted at the start of *ILF* into the module `tactics`. These files can also include other commands to be performed, notably commands like `?- load("Filename.tac").` to reconsult more tactic files.

In principle, a tactic is developed and tested like any other programme in PROLOG-2 (see the PROLOG-2 manual for details). Basic tactics are provided by the integrated deductive systems as rules of inference or as commands for moving a focus to a certain position in the proof. From these, more complex tactics can be build using all the facilities of PROLOG 2 and some special tools described below. As long as tactics manipulate proofs only through the use of rules of inference, the correctness of the resulting proofs is guaranteed.

The specific set of commands that are available for building proof tactics depends on the deductive system in use. However, the following commands have been implemented for many systems in order to unify their use.

common tactics commands

`pos(X)`	unifies `X` with the current position in the proof
`set_pos(Pos)`	moves the focus to the position `Pos`
`successors(S)`	unifies `S` with the list of successors of the current position. This assumes that the proof is represented as a directed graph.
`predecessors(S)`	unifies `S` with the list of predecessors of the current position. This assumes that the proof is represented as a directed graph.

`subgoals(S)`	unifies `S` with the list of open subgoals that have to be solved in order to solve the goal at the current position. The exact meaning of the concept of an open subgoal depends on the specific deductive system in use
`fd(N)`	moves the focus to the `N`-th successor of the current position
`bd(N)`	moves the focus to the `N`-th predecessor of the current position
`up`	is the same as `fd(1)`. Backtracking does not undo the effects of the positioning commands. Therefore, in order to continue editing a proof at a certain position after performing a tactic, this position has to be stored before the tactic and restored after the tactic has finished by a construction like `pos(X),Tactic,set_pos(X)`
`d`	is the same as `bd(1)`
`problem(Pos,Con,Stat)`	matches the formula at the position `Pos` in the proof with `Con` and the status of the position (e.g. proved or unproved) with `Stat`. The exact meaning of contents and status is system dependent. `problem/3` is often used to analyze the current goal in order to chose an appropriate tactic

In many systems the application of a rule of inference is triggered by specifying a certain axiom. In *ILF* a name `Name` of an axiom can be used to call a procedure `ax(Name)` that acts in a system specific way. On backtracking, the effect of this procedure is undone.

The following *tacticals* can combine tactics. They are similar to the corresponding tacticals of the *Oyster* system ([BH90]).

common tactics commands

`repeat_tac(Tac)`	applies the tactic `Tac` at the current position and recursively to all open subgoals generated by `Tac`. This is done in a depth-first/left-to-right way. Backtracking into `Tactic` is possible
`Tac1 then_tac Tac2`	applies the tactic `Tac1` at the current position and `Tac2` at all open subgoals created by `Tac1`. Backtracking into `Tac2` is possible; if this has failed on all subgoals, backtracking into `Tac1` is performed
`Tac then_l [T1,...,Tn]`	succeeds if the tactic `Tac` generates exactly `n` open subgoals to which `T1,...,Tn` are applied successfully. Backtracking into `Tn,...,T1,Tac` is possible

Global variables and loops are available for a procedural style of programming.

global variables commands

`assign(R(A1,...,An),V)`	stores `V` in `R` in association with the parameters `A1,...,An` (`n` can be 0). For all settings of the parameters `A1,...,An` at most one value `V` is stored in `R`. It can be retrieved by `R(A1,...,An,V)`
`R(A1,...,An) := V`	evaluates `V` to a number and stores the resulting value in `R` in association with the parameters `A1,...,An`. The value of `V` can be retrieved by `R(A1,...,An,Value)`
`(Procedure until Test)`	applies `Procedure` until `Test` succeeds. `Test` is called after each run of `Procedure`. No backtracking into `Procedure` is performed
`(Procedure while Test)`	applies `Procedure` as long as `Test` succeeds. `Test` is called before each run of `Procedure`. No backtracking into `Procedure` is performed

`Example:`

```
my_status_tac(T,S) :- pos(P),(T until problem(P,_,S)).
```

`my_status_tac(T,S)` applies the tactic `T` until the initial position `P` has a status that matches `S`.

A complex tactic may generate large new subproofs. It may not be possible for a tactic to determine the size of these proofs in advance. In such a situation it can be useful to forget those positions that have been created by the tactic except for those which contain new open subgoals that still have to be proved. This can be achieved by running the tactics hard. In order to do this, call `Tactic h_d`. Since a part of the result of the tactic has been eliminated, backtracking into `Tactic` is disabled. It appears as if a single new rule of inference that generated the new subgoals has been called.

Chapter 7

Background Experts

Background experts are systems controlled by the PROLOG running as a background process. Each of these experts can occur in several instances working on completely separated data. In order to use an expert, it must be mentioned in the `.ilfrc` file read at startup.

An expert named `expert` is started by `ex_start(expert,Nr)`. If this call succeeds, `Nr` is instantiated with the number assigned to this expert by the internal ExpertManager. Commands can be sent to this expert by using the syntax `Nr : Command`$_1, \ldots,$`Command`$_n$. If `Nr` is 0, the commands are sent to the background ExpertManager.

If the expert has been successfully started, it must be usually configured by reconsulting an appropriate file from its directory `${USERILFHOME}/dedsys/expert`. This can be done automatically if a file has been mentioned in the `.ilfrc` file or from the command line or by a tactic using the command `Nr : open_sit(File)`. The internal situation of an expert can be saved by `Nr : save_sit(File)`.

In order to use a background expert it is important to synchronize the work of the foreground deductive system and the background expert. Communication is synchronised by the command `ex_sync/0`. All axioms having a name matching `Name` are sent to expert `Nr` by `Nr : send_ax(Name)`. Axioms can also be activated and deactivated for selected experts as for the foreground system. The easiest way to synchronize the knowledge bases of the foreground and a background expert is by using the commands `ax_sync/1, th_sync/1 and kb_sync/1`. `ax_sync(Nr)` assures that the foreground and the background expert `Nr` have the same axioms with the same names; `th_sync/1` synchronizes the hierarchies of theories and `kb_sync/1` performs both.

Before a background expert can be asked, an expert goal must be built. This can be done interactively by `Nr : get_expert_goal/0` or automatically by a tactic. For the latter method, `make_exp_goal(Goal,expert,unproved,Name,Nr)` should be called. If this call succeeds, the background has accepted to try to prove `Goal`. `Name` can be an arbitrary PROLOG term that will be passed to the background system and back. It can be used to identify the results of the proof or to pass further control information to the background system.

When the background system has started its work on the goal, there will be a fact `expert_goal(Goal,expert,started(Job),Name,Module)` asserted in the foreground. Here, `Job` is the number of the job in the background and `Module` is the current input/output module when the problem was set up. By reference to the module the ExpertManager can continue to handle a job even if the deductive system which started the job has been suspended for some time or if different deductive systems start jobs with the same name. When the job gets done, `started(Job)` is changed to `finished(Job)`. If the goal has been proved, a fact `expert_goal(Goal,expert,proved,Name,Module)` will be present. Expert goals can be removed by `rm_exp_goal/5`. No deductive system is allowed to remove expert goals initiated by other systems.

In order to obtain the results of the background systems, the foreground system must call `ex_answer/0` from time to time.

In general it is impossible for a background expert to decide whether it should keep on trying to find a proof or better try another problem. In order to make this decision in an automated way the method of *flexible killing* can be used to interrupt the work of automated experts running as separate processes. A maximum and minimum number of seconds a system is allowed to run can be set up by `Nr : set_max_sec(Seconds)` and `Nr : set_flexy_sec(Seconds)`. Default values can be determined in the *ilf_state*s `default_max_sec` and `default_flexy_sec`, respectively. Note that the time given in `default_max_sec` is CPU time, but the time in `default_flexy_sec` is real time. If a command `kill_flexy/0` is send to the expert `Nr`, its processes will be killed if its minimum number of seconds has expired. If the time given in `default_max_sec` is reached, the processes belonging to the expert will be killed by the cpu limit mechanism of the C shell.

Chapter 8

Deductive Systems

We distinguish interactive and automated deductive systems. Usually, interactive systems are controlled by the PROLOG working in the foreground, while automated deductive systems work in the background, sometimes as separate processes.

8.1 Interactive Deductive Systems

8.1.1 FLEX – SLD Resolution

FLEX is an interactive proof editor for SLD-resolution proofs for universal Horn theories. A (partial) proof is given as a tree of literals with a unique root `input` which stands for contradiction. The semantics of such a proof tree is given as follows: The universal closure of a node is proved if the universal closure of all of its sons is proved.

`ax/1` is the only inference rule of *FLEX*. It applies the axiom whose name is given as its argument to the actual position, i.e. if the actual position is a leaf and the literal at this position matches the head of the axiom, then the most general unifier is applied to the tree and the literals of the body of the axiom are added as sons of the current position.

If the axiom is a fact, the current node is treated as being closed. A tree having only closed leaves is a proof. The name given as argument to `ax/1` can be any valid PROLOG term including _. On backtracking all axioms with matching names will be applied to the actual position. For the user's convenience for every axiom named `Name` a clause `Name :- ax(Name)` is asserted, so axioms can be called by simply typing their names.

For positioning in the proof tree the commands `d/1`, `d/0`, `up/1`, `up/0` , `up_down/2` described in Section 6 can be used.

`last/0` positions on the most recently generated leaf.

`tree/0` shows the actual proof tree.

`tree(N)` shows the actual proof tree starting N nodes above the actual position.

`problem/0` shows the literal at the current position together with its state (`proved` or `unproved`, `leaf` or `node`).

`h/0` shows the actual possibilities to proceed, that is, the open leaves that can be selected or the axioms appliable to the selected leaf.

`back/0` does explicit backtracking.

A sequence of positioning commands and inference rule applications in *FLEX* can be recorded as a tactic by `record_tac/0`. The tactic recorder is stopped by `end_tac`. The user will be prompted for a name under which the tactic will be saved. A saved tactic can be stored in a file using `store_tac/0`.

FLEX can be started by `flex/0` or `flex/1`. If no tactic is given as an argument, the tactic `break` is used and *FLEX* will work interactively. If it is not started by the command `Head to Body by flex(Tactic)`but simply by the command `flex`, *FLEX* prompts the user for a goal to prove which must be a conjunction of literals. The proof tree is initialized by adding the literals of the goal as sons of the root.

If no tactic is called *FLEX* works in PROLOG-2's break modus, which can be left by typing `end_of_file` or selecting `leave FLEX` from the `flex` menu (in fact, this is currently the only point to select from this menu).

Successors of a node in the proof tree are its sons and the only predecessor is its father. Subgoals of a node are the leaves below it that are not closed. These concepts are relevant for the use of tacticals (see Section 6).

8.1.2 ME – Model Elimination

ME is an interactive proof editor for the model elimination procedure as suggested by Stickel in [St84]. Stickel considered the model elimination as an extension of PROLOG–style logic programming from Horn clauses to arbitrary clauses. Therefore *ME* was implemented as an extension of *FLEX*, so most of the functionality of *FLEX* is also available under *ME*. The code both systems are sharing is located in `kern.prm` while the system–specific code is in `me.prm`.

A (partial) proof is a tree of literals with the root `false`, which stands for contradiction. The semantics of this tree are given similiar to *FLEX*: if the universal closure of all sons of a given node is proved, then the universal closure of this node is proved, too. A tree whose leaves are proved is therefore a proof of a contradiction, i. e. a refutation for the given set of clauses. Completeness for first order Horn–logic is achieved by using all contrapositives of all clauses including the goal and an additional inference rule, the (ME–)reduction rule (the inference rule of PROLOG or *FLEX* is in this context called extension rule).

ME can be started by `me/0` or `me/1`. If no tactic is given as argument, the tactic `break` is used, i. e. *ME* works interactively. If necessary, it prompts the user for a goal to prove which must be a conjunction of literals. The proof tree is initialized by adding these literals as sons of the root.

Because of the need of all contrapositives *ME* uses its own theory data base formed during startup. If you add axioms to *ILF*'s theory while using *ME*, `use_cp/0` must be called to make the contrapositions available for *ME*. The name of a contrapositive is derived from the name of the corresponding axiom by adding the number of the literal considered to be the head as last argument (e. .g. from `ass(+)` you will get `ass(+,1)`, `ass(+,2)`, ... corresponding to the choice of the first, second, ... literal as the head). The names of the contrapositives of the negated goal are `goal(1)`, `goal(2)`, ...

All contrapositives can be listed by `cps/0`.

`ax/1` is the command for the extension rule and works as in *FLEX*. For the user's convenience for every contrapositive/axiom named `name` a clause `name :- ax(name)` is asserted as in *FLEX*.

The reduction rule can be applied via the `red/1` command. `red(N)` reduces the actual leaf with its `N`–th ancestor, i. e. if the current goal matches the complement of its `N`–th ancestor goal, then the most general unifier will be applied to the tree and the current goal will be treated as solved. If `N` is not instantiated, `red/1` finds all possibilities for ME–reduction by backtracking. `red/0` is a shorthand for `red(_)`.

The tree positioning commands and the concepts of successors, predecessors and subgoals are the same as in *FLEX* (see Section 8.1.1).

8.1.3 MPRT

MPRT is an interactive proof editor for the modified problem reduction format as defined by Plaisted in [Pl88].

Proofs are represented as trees similiar to *FLEX* and *ME*. A discussion of the tree representation of modified problem reduction can be found in [Me90].

MPRT uses three inference rules: axiom application, assumption application and splitting (see below). Note that *MPRT* is much simpler than the *TMPR*–prover developed by Mellouli[1].

The system consists of two parts: the executable `mprt` connected with *ILF* as described in Section 9 and the Prolog module `mprt.prm` which provides some additional functionality. The executable was developed in the programming language C using the *ILFA*–libraries (see [DFKBLL]).

`ax/1` is the command for the axiom application rule. It applies the axiom whose name is given as an argument at the actual position, i. e. if the actual position is a leaf and the literal at this position matches the head of the axiom, then the most general unifier is applied to the tree and the literals of the body of the axiom are added as sons of the current position. If the axiom is a fact, then the current node is treated as proved. The name given as argument to `ax/1` can be any valid Prolog term including _. On backtracking all possible axioms will be tried at the actual position. For every axiom named `name` a clause `name :- ax(name).` is asserted for the user's convenience.

`ass/1` applies an assumption at the actual position, i. e. if the actual position is an unproved leaf and the assumption identified by the argument matches the literal

[1] *MPRT* is comparable to the *WMPR* (weak MPR) described in [Me90].

of the leaf, the matching substitution is applied to the tree and the current node is marked as proved by assumption.

`case/1` splits the current node over the literal provided as argument. This rule is applicable to each node of the tree, not only to leaves. The positioning commands described in Section 6 are available.

`problem/0` shows the literal at the current position together with its state (`proved` or `unproved`, `leaf` or `node`). By `problem/3` you can use this information in a tactic.

`h/0` shows the axioms and assumptions applieable at the current node. Under the graphic user interface a menu `Rules` containing these possibilities is constructed. `h` is accessible via the `help` point from this menu.

`back/0` does explicit backtracking.

MPRT is started by `mprt/0` or `mprt/1`. If no tactic is given as the argument, the tactic `break` is used, i. e. *MPRT* works interactively.

The core of *MPRT* is a C programme. Under the graphic user interface a window for error messages from this programme will be created presenting the process id of the executable in the title. *MPRT* has to be initialized by `mprt_start/0`.

The proof tree is initialized by applying the goal to the root and positioning to the first son of root.

Interactive work in *MPRT* can be left by typing `end_of_file` or selecting `leave` from the `mprt` menu.

8.1.4 pNAT – PROLOG Based Natural Reasoning

The *pNAT* is an Interactive Theorem Prover on the calculus of Natural Reasoning, as it is usually described in the literature. A detailed description can be found in [Wo92].

Commands of pNAT

`sequents.`	lists out all sequents of the actual situation of the system
`nat_start.`	initializes or reinitializes the system, all previous information will be lost
`load_theory.`	converts all *ax_name* facts from the current situation module into the internal syntax of pNAT
`load_goal.`	converts the *ilf_goal* into the internal syntax of pNAT

`use(Rule,Info,Pos_list).`	performs a deduction step in the calculus, predicate is backtrackable, `Rule` can be one of the following: *in(not)*, *out(not)*, *in(and)*, *out(and_1)*, *out(and_2)*, *in(or_1)*, *in(or_2)*, *out(or)*, *in(imp)*, *out(imp)*, *in(aeq)*, *out(aeq_1)*, *out(aeq_2)*, *in(all)*, *out(all)*, *in(ex)*, *out(ex)*, *ns1* or *ns2*, `Info` is in most of the cases a pattern of the right side to be built and may often be a variable, and `Pos_list` is the List of parent sequents, often all parameters can be variables
`use(Offs,Rl,Info,Posl).`	as `use/3`, the Value of `Offs` will be added to the numbers in the `Posl`, can be used to access to relatively adressed sequents, `Offs` and `Posl` may not be Variables, else the predicate fails
`control(Left,Right).`	is true if there is a sequent of the given form
`result(Pos).`	asserts the sequent with the highest position (number) as proved to an *ilf_goal*
`offset(Nr).`	`Nr` will be the number of the sequent created last, it is usable e.g. with `use/4`
`set_pos(Pos).`	sets the actual position of the system to be `Pos`, backtrackable
`pos(Pos).`	gives back as `Pos` the actual position, backtrackable
`predecessor(Pred).`	gives back as `Pred` a parent of the actual position, backtrackable
`predecessors(Predl).`	gives back as `Predl` the list of all parents of the actual position
`predecessors(Pos,Predl).`	gives back as `Predl` the list of all parents of `Pos`
`successor(Succ).`	gives back as `Succ` a child of the actual position, backtrackable
`successors(Succl).`	gives back as `Succl` the list of all children of the actual position
`successors(Pos,Succl).`	gives back as `Succl` the list of all parents of `Pos`
`up(N).`	sets the actual position to be the `N`–th predecessor of the current actual position
`down(N).`	sets the actual position to be the `N`–th successor of the current actual position
`contents(Seq).`	gives back as `Seq` the contents of the actual position
`contents(Pos,Seq).`	gives back as `Seq` the contents of the position `Pos`
`status(Stat).`	shows the status of the actual position, `Stat` can be *initial* (the root node), *axiom*, *leaf* or *node*

`status(Pos,Stat).`	gives back as `Stat` the status of the position `Pos`
`back.`	retracts the sequent created last
`back(N).`	retracts the sequent with the number `N`, it naturally must be a leaf

The `subgoal.` command is not implemented, because it makes no sense in the context of the *pNAT* system.

8.1.5 pTAB – PROLOG Based Tableau Calculus

The *pTAB* is an interactive theorem prover based on the tableaux calculus, as it is usually described in the literature (see [Sm68]). For details of this implementation see [Wo92].

Commands of pTAB

`nodes.`	lists out all nodes of the tableau tree
`structures.`	lists out all variable structures of the current tree (see implementation)
`show_branch(Leaf).`	shows all nodes on the path from `leaf` to the root node of the tableau tree
`show_tree.`	shows all branches
`proof.`	tests if the current tree is a proof, too
`tab_start.`	(re-) initializes the prover
`load_theory.`	integrates the current knowledge base in the tableau tree, new axioms will be built in as root node
`load_goal.`	integrates the current *ilf_goal* into the tree
`set_pos(Name,Leaf).`	sets the actual position to be on the node `Name` and on the path above `Leaf`
`set_pos([Name,Leaf]).`	as `set_pos(Name,Leaf).`
`successor(X).`	gives back a successor of the actual position, backtrackable
`predecessor(X).`	gives back a predecessor of the actual position, backtrackable
`top.`	sets the actual position on the root node and an arbitrarily selected leaf
`up.`	sets the actual position on the predecessor
`up(N).`	`N` times *up*.
`d.`	sets the actual position on the first sucessor
`d(N).`	sets the actual position on the `N`-th sucessor, corrects the actual leaf

`left.`	sets the actual position on the left brother of the old position
`left(N).`	sets the actual position on the `N`-th left brother of the old position
`leaf(Leaf).`	gives back as `Leaf` a leaf of the tableau tree
`leaves(X).`	gives back the list of all leaves
`open_branch(Leaf).`	gives back as `Leaf` an open leaf of the tableau tree
`open_branches(X).`	gives back the list of all open leaves
`subgoal(X).`	gives back an open leaf below the actual position, backtrackable
`contents(Node,Formula).`	gives back as `Formula` the formula of the node `Node`
`status(Name,Leaf,Sta).`	gives back as `Sta` the status informations of the node `Node` with respect to the leaf `Leaf`
`split(Name,Leaf).`	splits the formula of the node `Name`, and expands the tree by appending the resulting formulas of the split below the leaf `Leaf` (if all that is possible), the predicate is backtrackable, the arguments may be variables
`close_branch(Leaf,N).`	tries to close up the branch above `Leaf`, if `N` is an integer, one of the two nodes used for a closure must have a node number greater than `N`
`close_branch(Leaf).`	the same as `close_branch(Leaf,_).`
`close_branch.`	prompts for a leaf and (re-) tries to close up the branch belonging to that leaf
`back.`	prompts for a node and takes back all splits done after the creation of this node

8.1.6 iTAB – Tableau Calculus based on the ILFA Library

The *iTAB* is an interactive theorem prover based on the tableaux calculus, as it is usually described in the literature (see [Sm68]), implemented in C using the libraries of the *ILFA* system (see [DFKBLL]).

Commands of itab

`back.`	takes back the last tree expanding command
`load_theory.`	transmits the current knowledge base to *itab*
`goal.`	transmits the current *ilf_goal* to *itab*

`local_pos([Node,Leaf]).`	reads the list of local positions, i.e. the actual node and the actual leaf from *itab* and assigns that information to be the argument of the predicate *local_positions* (as a list)
`set_local_pos([Nd,Lf]).`	sets the the actual node and the actual leaf to be `Nd` and `Lf`
`pos(Node).`	reads the actual position from *itab* and assigns that information as the argument of the predicate *actual_position*
`set_pos(Node).`	sets the the actual node to be `Node`
`contents(Cont,Var).`	reads the formula contained in the node of the actual position from *itab* as `Cont` with the variable name structure `Var`
`contents(Cont).`	the same as `contents(Cont,_).`
`contents(Pos,Cont,Var).`	as `contents(Cont,Var).`, but not of the actual position but of the node `Pos`
`status(Stat).`	gives back as `status` the status information of the actual node in relation to the actual leaf, i.e. if the branch is closed or open
`status(Pos,Stat).`	as `status(Stat).`, but not of the actual position but of the node `Pos`
`subgoals(Sub).`	gives back as `Sub` the list of all open leaves below the actual node
`subgoals(Pos,Sub).`	as `subgoals(Sub).`, but not of the actual position but of the node `Pos`
`successors(Succ).`	gives back as `Succ` the list of all nodes directly below the actual node
`successors(Pos,Succ).`	as `successors(Succ).`, but not of the actual position but of the node `Pos`
`predecessors(Pred).`	gives back as `Pred` the list with the node directly above the actual node
`predecessors(Pos,Pred).`	as `predecessors(Pred).`, but not of the actual position but of the node `Pos`
`fd(N).`	sets the actual node to be the `N`th successor of the current actual node, warns if the leaf is not set correctly
`bd(1).`	sets the actual node to be the predecessor of the current actual node
`set_leaf(Leaf).`	sets the actual leaf to be `Leaf`
`ax(Name).`	splits the axiom `Name` below the actual leaf, if possible
`branch.`	shows the contents and status of all nodes above the actual leaf with respect to that leaf

`s(Node,Leaf).`	tries to split the node `Node` and to expand the tree below the leaf `Leaf`
`c(Leaf).`	tries to close up the branch above the leaf `Leaf`

8.2 Automated Deductive Systems

8.2.1 DISCOUNT

This system integrates the *DISCOUNT* system for distributed knowledge based equational reasoning and completion from the University Kaiserslautern [DP92]. The flags and parameters follow the user manual of *DISCOUNT*. The system automatically works in the silent and team mode and will perform *PCL* with the same input files if the task can be proved to get the dependencies. It is possible to configure experts, referees and selectors (in the sense of *DISCOUNT*) and to combine them to teams. Unlike the original *DISCOUNT*, experts, referees and selectors have to be referred to by names that can be arbitrary PROLOG terms.

DISCOUNT menu

`ask`	n	gives the current *expert_goal* to the background and expert starts working, performs `ex_ask`.
`end`	n	terminates this expert, performs `ex_end`.
`refresh`	n	rebuilds the menu with the current menu level, performs `refresh_menu`.
`send_axioms`	c	transmits the matching axioms of the current knowledge base to the expert, performs `send_axioms(Name)`.
`goal`	c	prompts the user to type a formula to use it as an *expert_goal*, performs `get_expert_goal`.
`use result`	c	prompts the user for a name to integrate a proved *expert_goal* with the given name into the current knowledge base, performs `use_expert_goal`.
`kill_flexy`	n	kills the actual task of the expert if the minimum time (*default_flexy_sec(Sec)*, set in the .ilfrc file) for the task is over, performs `kill_flexy`.
`file`	s	submenu relating to file operations
`open_sit`	c	performs `open_sit(File).`, prompts the user for the name of the file with the data module to be used by the expert
`save_sit`	c	performs `save_sit(File).`, prompts the user for the name of the file to save the data module as this file

In higher menu levels there are many other menu points to configure the team of *DISCOUNT* experts, here are listed only the points that are additions to the user manual.

more from the DISCOUNT menu

`set_flexy_sec`	c	prompts the user to type in the minimum (real) time *DISCOUNT* can work without getting killed, performs `set_flexy_sec(Sec).`
`max_seconds`	c	prompts the user to type in the maximum (cpu) time *DISCOUNT* can work without getting killed, performs `set_max_sec(Sec).`
`define experts` → `build` → `statistics`		
	s	nested submenu to create and name an expert, here a *statistic* expert, other types see the menus and the user manual of *DISCOUNT*
`add_weight`	c	the user shall fill in the pattern *bld(exp, Name, add_weight(FWeight, VWeight, RedInst, Order))* with the wanted parameters to define the *add_weight* expert *Name*
`define spec` → `build`		
	s	nested submenu to create and name a specialist
`reduce_cp`	c	the user shall fill in the pattern *bld(spec, Name, reduce_cp(StrtCp, RdCp, SsCp, DblCp))* with the wanted parameters to define the *reduce_cp* specialist *Name*
`define referee` → `build`		
	s	nested submenu to create and name a referee
`statistic`	c	the user shall fill in the pattern *bld(ref, Name, statistic(SR, SE, SCP, RedCnt, MsFg))* with the wanted parameters to define the *statistic* referee *Name*
`define select` → `build`		
	s	nested submenu to create and name a selector
`statistic`	c	the user shall fill in the pattern *bld(sel, Name, statistic(NrRls, NrEqs, MinV, RCnt, RRt, RLt, REq, RGl, RSsum, CpCnt))* with the wanted parameters to define the *statistic* selector *Name*
`team` → `static`		
	s	nested submenu to create and name a *static* triple of expert, referee and selector
`add`	c	the user shall fill in the pattern *add_er(static, Exp, Ref, Sel)* with the names of the predefined expert, referee and selector, the system will treat them as a triple for the configuration of *DISCOUNT*

8.2.2 SETHEO

The *SETHEO* system uses the *SETHEO* resolution based automated theorem prover from the Technical University Munich [LSBB92]. The system will verify the dependencies of the goal from the input formulas and write them back to the user.

SETHEO menu

`ask`	n	gives the current *expert_goal* to the background and expert starts working, performs `ex_ask.`
`end`	n	terminates this expert, performs `ex_end.`
`refresh`	n	rebuilds the menu with the current menu level, performs `refresh_menu.`
`send_axioms`	c	transmits the matching axioms of the current knowledge base to the expert, performs `send_axioms(Name).`
`goal`	c	prompts the user to type a formula to use it as an *expert_goal*, performs `get_expert_goal.`
`use result`	c	prompts the user for a name to integrate a proved *expert_goal* with the given name into the current knowledge base, performs `use_expert_goal.`
`kill_flexy`	n	kills the actual task of the expert if the minimum time (*default_flexy_sec(Sec)*, set in the .ilfrc file) for the task is over, performs `kill_flexy.`
`set_flexy_sec`	c	prompts the user to type in the minimum (real) time *DISCOUNT* can work without getting killed, performs `set_flexy_sec(Sec).`
`max_seconds`	n	prompts the user to type in the maximum (cpu) time *DISCOUNT* can work without getting killed, performs `set_max_sec(Sec).`
`file`	s	submenu relating to file operations
`open_sit`	c	performs `open_sit(File).`, prompts the user for the name of the file with the data module to be used by the expert
`save_sit`	c	performs `save_sit(File).`, prompts the user for the name of the file to save the data module as this file

8.2.3 OTTER

OTTER is an automated theorem prover from Argonne National Laboratory ([MC90]) for first order theories with special support for reasoning about equality. The syntax of the commands is as described in *Otter*'s manual, except that theories and goals are taken from *ILF*. Therefore, we describe only the commands to control the work of the expert.

OTTER menu

`ask`	n	gives the current *expert_goal* to the background and expert starts working, performs `ex_ask.`
`end`	n	terminates this expert, performs `ex_end.`
`refresh`	n	rebuilds the menu with the current menu level, performs `refresh_menu.`
`send_axioms`	c	transmits the matching axioms of the current knowledge base to the expert, performs `send_axioms(Name).`
`goal`	c	prompts the user to type a formula to use it as an *expert_goal*, performs `get_expert_goal.`
`use result`	c	prompts the user for a name to integrate a proved *expert_goal* with the given name into the current knowledge base, performs `use_expert_goal.`
`kill_flexy`	n	kills the actual task of the expert if the minimum time (*default_flexy_sec(Sec)*, set in the .ilfrc file) for the task is over, performs `kill_flexy.`
`file`	s	submenu relating to file operations
`open_sit`	c	performs `open_sit(File).`, prompts the user for the name of the file with the data module to be used by the expert
`save_sit`	c	performs `save_sit(File).`, prompts the user for the name of the file to save the data module as this file
`otter_input`	c	prompts the user for the name of the file to use as *OTTER* input file, performs `otter_input(File).`

8.2.4 An AC Rewrite System for Lattices and Groups

The *BgPest* system tries to prove a given formula by rewriting due to a method of Peterson and Stickel [PSt81]. It decides automatically whether the theory of groups or of lattices is to be used, depending on the leading operator of the left side of the equation. If there is no operator, then one will be searched on the right side.

The result will be the (lexicographically smallest) normal forms of both sides of the equation, but only if these sides are not variables. The usage follows the standard of the ExpertManager, but it is not necessary and not possible to send theories to the background.

BgPest menu

`ask`	n	gives the current *expert_goal* to the background and expert starts working, performs `ex_ask.`
`end`	n	terminates this expert, performs `ex_end.`

`goal`	c	prompts the user to type a formula to use it as an *expert_goal*, performs `get_expert_goal`.
`use result`	c	prompts the user for a name to integrate a proved *expert_goal* with the given name into the current knowledge base, performs `use_expert_goal`.

8.2.5 Algebra

algebra is an automated deductive system specialized to utilize simple algebraic properties. Unlike other automated systems it works in the foreground because it performs relatively simple actions.

When *algebra* is started for the first time it checks the actual knowledge base for axioms saying that some of the operators are associative, commutative, or distributive. Currently, *algebra* has the following rules.

algebra menu

`iso_rule(P).`	reduces a goal `G` with subterm `T` at position `P` to `(T=U,G')`, where `U` is a new variable and `G'` is obtained from `G` by replacing `T` at position `P` by `U`
`demod_rule(Name,PosList).`	applies an equation name `Name` at the positions in the list `PosList` of the goal. If the equation has preconditions, they are collected in the body of the goal to be proved
`dis_rule(P).`	reduces a goal `G` by applying a distributive law at position `P` of `G`. If the same law can be applied at the newly generated subterms, this is performed
`ac_move_rule(Src,Dest).`	reduces a goal `G` by moving the subterm at position `Src` to the subterm at position `Dest`, provided this is justified by associative and commutative laws

8.2.6 TwoLat – the Two Element Lattice

The *TwoLat* system is not a deductive system in the usual sense, it tests formulas in the lattice with two elements. The fact that a universal Horn formula is true in that model, can be treated as a proof of this formula from the theory of distributive lattices (see [Da93]). Moreover, the system can be used as a model tester to prevent proofs of formulas not valid in that model (so-called junk theorems). The usage follows the standards of the *ExpertManager*, but it is not nessecary and not possible to send theories to the background. The user must recall, that "proved by *TwoLat*" means only valid in the model of *TwoLat*.

TwoLat menu

`ask`	n	gives the current *expert_goal* to the background and expert starts working, performs `ex_ask`.
`end`	n	terminates this expert, performs `ex_end`.
`goal`	c	prompts the user to type a formula to use it as an *expert_goal*, performs `get_expert_goal`.
`use result`	c	prompts the user for a name to integrate a proved *expert_goal* with the given name into the current knowledge base, performs `use_expert_goal`.

8.2.7 ThreeLat – a Three Element Lattice Model

ThreeLat is a system that checks the validity of a quantifier free formula G from the language of lattice ordered groups in the integers. G may contain additional constants, and facts about these constants may be given as additional axioms. More precisely, it is checked whether G holds in this model for all instances of the constants with the values $-1, 0, 1$ satisfying the additional axioms. If this is true and G is a Horn formula which does not contain the symbol for the group operation, *ThreeLat* returns `proved` since G is a logical consequence of the theory of lattice ordered groups and the additional axioms (see [Da93]). If G holds but is not a Horn formula or contains the group operation symbol, *ThreeLat* returns `passed`, otherwise it returns `refused`. The usage follows the standard of the ExpertManager, but it is not necessary and not possible to send theories to the background.

ThreeLat menu

`ask`	n	gives the current *expert_goal* to the background and expert starts working, performs `ex_ask`.
`end`	n	terminates this expert, performs `ex_end`.
`goal`	c	prompts the user to type a formula to use it as an *expert_goal*, performs `get_expert_goal`.
`use result`	c	prompts the user for a name to integrate a proved *expert_goal* with the given name into the current knowledge base, performs `use_expert_goal`.

Chapter 9

Extending ILF

Extending the power of *ILF* requires more information on the internal structure of *ILF*, which will be given in this section. We shall describe how to integrate new interactive and automated systems.

9.1 Integration of Interactive Systems

The knowledge base of *ILF* is kept in the PROLOG module `axioms.prm` in the predicate `ax_name/4`. The first argument of this predicate is the name of the axiom, the second and third argument are head and body of the sequent constituting the axiom. The last argument is a variable name structure as described in the manual of PROLOG 2. If the structure of a sequent is not relevant, the complete formula can be contained in the second argument while the third argument is `true`.

When a new deductive system is called, the corresponding module is loaded and a separate module for the data of the system is created. If the system contains predicates defined elsewhere, the new definitions override the existing ones. This makes it possible to realize the same commands in different systems by different procedures. When a system finishes its work, the original definitions are restored.

Deductive systems are accessed via the deduction system manager, which is called by `ded_sys_man/2`. The first argument is the deductive system to activate, the second a tactic to execute.

By convention every deductive system *DedSys* provides the predicates `DedSys/0` and `DedSys/1` defined as

```
DedSys :- ded_sys_man(DedSys,break).
DedSys(Tactic) :- ded_sys_man(DedSys,Tactic).
```

If the deduction system manager gets messed up (for instance by leaving the `Tactic` or the `break` by abort) use `reset/0` to set it in its initial state.

When a deductive system starts, the deduction system manager has created a data module containing only a fact `ilf_goal(Head,Body,unproved,Name)` and being the current input/output module. `Body` $\longrightarrow$ `Head` is the formula to be proved. The last argument `Name` is a name for this formula if the system was called by the command `reduces` and `none` otherwise.

When the system is left, the third argument must be changed from `unproved` to `proved` if the goal has been proved. It is also possible to instantiate variables in the formula if only an special case of the goal could be proved.

The designer of a deductive system must be aware that *ILF* may backtrack into the system in order to look for another proof of the same goal.

A deductive system is integrated by defining a new fact:

```
ded_sys_props(System,Modules,CallProcedure,ExitProcedure)
```

Here

- `System` is the name of the system,
- `Module` is a list of PROLOG modules to be loaded,
- `CallProcedure` is a PROLOG predicate that is performed before any tactic,
- `ExitProcedure` is a PROLOG predicate that is performed before the deductive system is left, if the tactic succeeded.

A new interactive deductive system should – if possible – define the predicates `successors`, `predecessors`, `subgoals`, `problem` described in Section 6 in order to make tacticals available.

New deductive systems developed in C must also be connected with *ILF* through a PROLOG module.

Strictly speaking, the C programme required is a collection of procedures. Some of these procedures are necessary. E.g. `load_theory` to load a knowledge base and return handles for the axioms read, `goal` to read a goal to be proved etc. A complete list with a description of the interface is contained in the file `ceditor.dok`. The `main()` function and the functions necessary to communicate with *ILF* are defined in `editor.c`. The C-programmer has to include `ceditor.h` and `editcmds.h` and link his object files with `editor.o` and `editcmds.o`. All these files are located in the `$ILFHOME/src/c/ceditor` directory.

The PROLOG side of the interface can make use of the modules `ilfa.prm` and `editor.prm`, which implement the standard commands and the predicates for the communication. If the call procedure of the new deductive system calls `ilfasys_init(Path,Title)`, where `Path` is the complete path to the executable programme, `Title` in connection with a process ID is used as the title of an X-term showing the `stderr` of the C programme.

9.2 Integration of Automated Theorem Provers

Automated theorem provers are usually integrated as background experts. They should have a PROLOG module `SYSTEM.prm` that has all its predicates private except for `SYSTEM_top/0`, which is called starting the system named `SYSTEM` and a special predicate `port_SYSTEM/1`, which must be defined by `port_SYSTEM(X) :- X`. Moreover, there must be the predicate `listen_problem/1`. If this predicate is called with an argument of the form `expert_goal(Goal,SYSTEM,Job,Name,Nr)`, the automated theorem prover should try to prove `Goal` from the theory it finds in its `ax_name/4` predicate in its data module `exp_Nr`, where `Nr` is the the number of the expert, obtained from the ExpertManager. Since all predicates in the data module of the system are private, different copies of the same system may be active at the same time. In order to access the data in the data module, `port/1` is used. If predicates from the data module are called by external predicates like `bagof`, they must be included in `port/1`. When the prover has finished its work successfully, it should call `output_write(Job,expert_goal(Goal,SYSTEM,proved(Job),Name,N))`. Finally it must call `output_write(Job,ready)`.

If an external system is not implemented within the PROLOG-2 in the background is called, `listen_problem/1` must produce an input file for the external system and start it as a background process using PROLOG's `call/1` command. Then `listen_problem/1` should succeed without the call of `output_write(Job,ready)`. For such systems there must be a predicate `search_results/3` defined for the system. The call `search_results(SYSTEM,Nr,Job).` informs the system `SYSTEM` with data module `exp_Nr` that the results of the Job `Job` can be analyzed. Usually, the raw output of the external system will be analyzed roughly by a filter programme which sends the call `search_results(SYSTEM,Nr,Job).` to the pipe `$USERILFHOME/tmp/coma.p`. The ExpertManager in the background prolog will forward this call to `SYSTEM`. `search_results` should finish calling `output_write(Job,ready)`.

It is recommended to start external processes with limited CPU time. After the call of the external system in the `listen_problem` predicate, a programme that finds out the PID's of the processes belonging to the external system should be called. These PID's should be sent to the pipe `$USERILFHOME/tmp/coma.p` as `pid(List)`, where `List` has the form `[Nr|PID_list]` and `PID_list` looks like `[PID1,...,PIDn]` or `[[Host1,PID1],...,[Hostn,PIDn]]`. Since the determination of the exact PID's is often not possible (e.g. using *DISCOUNT*), the flexy_kill mechanism will send signal 9 to the processes PID, PID+1 and PID+2.

Generic background experts can be found in the files `${ILFHOME}/doc/name.pro` and `${ILFHOME}/doc/name1.pro`.

9.3 Adding Menus

If the graphical user interface is active, a new menu named `MENU` can be brought up by the command `create_menu(MENU)`. The items of this menu are descibed by the predicate `menu_info_MENU/2` in the module `exmenus.prm`. The first argument of this predicate is a number `N` indicating that this item will be displayed at menu levels greater or equal to `N`.

If the item is a submenu, the second argument is the string to be shown as the title of the submenu.

By an *itemlist* we mean a list having three members. The first is one of the atoms `confirm` and `noconfirm`, indicating whether the action triggered by the menu item needs confirmation from the user or not. The second member is a string or an atom to be displayed in the menu and the last member is a string to be passed to PROLOG when the menu item has been selected.

It is also possible to create a separator within a menu by filling in `[separator]` as *itemlist*.

For menu items of the top level menu, the second argument of `menu_info_MENU/2` is an itemlist describing the item. For items of submenus the second argument is a difference between a sequence of `x` and an itemlist. The length of the `x`-sequence coincides with the depth of the item in the menu hierarchy.

Here we give an example of some possible lines:

```
menu_info_expert(0,[confirm,load_tac,"0:load_tac(Tac)"]).
menu_info_expert(0,[noconfirm,answer,ex_answer]).
menu_info_expert(0,[confirm,menu_level,"new_level(N)"]).
menu_info_expert(0,[separator]).
menu_info_expert(1,synchronize).
menu_info_expert(1,x-[noconfirm,communication,ex_sync]).
menu_info_expert(1,x-[confirm,axioms,"ax_sync(ExpertNr)"]).
```

Chapter 10

A Sample Configuration: The ProofPad

We describe a configuration of *ILF* that has been designed to make the power of recent deductive systems available to users not specially trained. The version currently implemented is configured to support proofs from the first order theory of lattice ordered groups. However, the description below will not refer to these special aspects. We shall explain the integration of the different parts of *ILF* within one application.

The *ProofPad* is a deductive system of very limited abilities that are considerably extended by the use of other deductive systems. It presents to the user as a sequence of proof lines to be edited. Each line has a status and the aim of the user is that each line has the status proved. However, the user cannot change the status directly. The system checks automatically, whether a specific line is a logical consequence of the preceding lines. It takes a fixed number of preceding lines into account. This number can be set with the *ilf_state* `pad_recall`; its default value is 3. Each time the user inserts a new line, the modified problems are passed automatically to a process running in the background. This background process tries to solve the problems within a limited amount of time. It can employ also other deductive systems, notably the automated theorem provers *SETHEO* and *DISCOUNT*. The work of these systems is controlled in a flexible way using the predicate `flexy_kill` described in section 7. If the background prover can prove a line, its status is changed in the foreground to `proved`, otherwise to `unproved`. Since the background can also test the validity of formulas in domain specific finite structures, it may also find out that a formula is unprovable and change the status accordingly to unprovable. This may happen because the user has made a semantic mistake or because the preceding lines do not yet contain enough information.

Like any other deductive system, it is called by `[Name reduces] Head to Body by pad(Modus)` (cf. 3.1). `Modus` can be `direct` or `indirect` depending on the type of proof to be edited. Currently only direct proofs are supported. It is also possible to call simply `pad`. Then the *ProofPad* will ask the user for the necessary information.

When the *ProofPad* has been started, the background is configured in an appropriate way. This can be seen by the menus of the background experts that become available now. The actual status of the proof can be inspected in three ways, which are all available either from the command line or from *ProofPads* pulldown menu.

viewer commands of ProofPad

`show_pad`	is the fastest way to see the proof in the command window
`pad_graph`	shows the structure of the proof in the graphics window of the *TreeViewer*. The single lines can be seen in the text window of the *TreeViewer*. The form of the presentation depends on the *ilf_state* `pad_mode`. If it is set to `line`, the proof is presented as a sequence of lines as by the `show_pad` command. However, if it is set to `tree`, the logical structure of the proof is shown as a directed graph. Edges of this graph indicate logical dependencies among the lines.
`pad_view`	this command produces a file `proof.tex` in the directory `${USERILFHOME}/tmp`. This is treated by LaTeX and presented by the `xdvi` command. Details on the LaTeX output are explained below.

In each of these presentations the goal will be named `Theorem` if the proof is complete. Otherwise it is displayed as `Conjecture`. The user may move through the lines using the commands `set_pos, up, d` described in Section 6. The command `last` takes him to the first line that has not been proved yet. The main tool of the user is the insertion of lines containing new formulas by the command `ins(Formula)` at the current position. These formulas get the status `untried`. Then they are automatically passed to the background system to be proved from a theory that can be set in the *ilf_state* `pad_default_theory`. If the background started working on the line, its status is changed to `tried` and can be changed later on by the background to `proved`, `unproved` or `unprovable`. `del` deletes the current line. The status of all lines that have been proved using the deleted line directly is changed to `untried`.

If the background is not able to prove the formula, the user can

- insert further lines or
- modify the theory to be used by the command `use(Theory)` or
- extend the theory by the command `use_also(Theory)`

Here `Theory` can be either the name of a theory or an axiom or a list of such names. Moreover, `Theory` can contain numbers of lines to be used, provided they occur before the current line. `pad_use_always(Theory)` specifies a theory that will always be added to the theories given to prove the lines on the *ProofPad*. Usually, this is applied to ensure that frequently needed axioms like the basic axioms of equality or of an ordering are not omitted.

The deductive system `algebra` (cf. Section 8.2.5) provides tools to construct new proof lines automatically. From the *ProofPad* they are used by the commands `ac_move` and `distribute`. These commands require parameters for the position of the subterms of the actual formula to be used. If they are called without parameters (e.g. from the pulldown menu of the *ProofPad*), the structure of the formula in the current line is displayed as a tree by the *TreeViewer* and the user can select the desired position with the right mouse button or he can display a subformula in the *TreeViewers* text window as a string and select a certain substring with the mouse. In the latter case, the position of the smallest subterm containing this substring is selected.

If an axiom is a Horn clause with a head matching the formula in the current line, the literals in the body can be automatically inserted as new lines in front of the current line by using the name of the axiom as a command. This mechanism is also used internally to connect the algebra system with the *ProofPad*. E.g. moving a subterm from `Pos1` to `Pos2` by `ac_move` calls the procedure:

```
pos(P),problem(P,F,_),
/* Getting the actual position and the contents of the current line (cf. section 6)*/
alge(dist) reduces F to F1 by algebra(ac_move_rule(Pos1,Pos2)),
/* Calling the algebra system (cf. section 3.1) with the rule of inference ac_move_rule. Note that F1 is a variable that will be instantiated by this call!*/
alge(dist),
/* Applying the axiom alge(dist) just created */
forget_ax(alge(dist))
/* Forgetting this axiom */
```

Lines of a proof can be reordered by the command `move_line(Line1,Line2)`, which will move `Line1` towards `Line2` as far as possible without violating the correctness of the proof. `move_lines_fd (move_lines_bd)` moves all lines to the bottom (to the top, respectively) as far as possible. This can provide a better readability of the proof, especially when transformed to LaTeX output.

The LaTeX output uses the article style of LaTeX. Author and title can be set by the *ilf_states* `pad_author` and `pad_title`, respectively. All files making up the LaTeX input reside in `${USERILFHOME}/tmp`. The files `proof.tex` contain the proof. The title is contained in the file `title.inp`.

For the LaTeX output, lines of equations or inequalities may be combined into a chain of equations or inequalities. These formulas are connected by the words "clearly", "hence", "therefore" and "by". "clearly" indicates that no other line in the proof has been used, "hence" means that just the formula immediately preceding the current formula has been used and "therefore" shows that just the number of immediately preceding formulas set in the *ilf_state* `pad_recall` have been used. All the other references are given explicitly. All axioms not belonging to one of the theories specified in the *ilf_states* `pad_default_theory` and `pad_use_always` are explicitly mentioned at the places where they have been used. All formulas that are added to the knowledge base at run time for further use with the *ProofPad* should be inserted in a theory specified in the *ilf_state* `pad_reference`. The default theory for this purpose is lemmata.

The *ProofPad* can be left by typing `end`. If a proof has been completed, the user has the possibility to write it into a LaTeX input file that can be used in a larger manuscript. In order to do this, the formula that has been proved must have a name `NAME`, which must be a PROLOG atom. Then the files `NAME.inp` - a LaTeX input file - and `NAME.dep` containing the relevant dependencies are created.

If the proof on the *ProofPad* is the last of a series of proofs of lemmata, a manuscript can be created. This is done in two steps. First, the command `outline` erases the *ProofPad* after saving the actual proof and builds a new proof by analyzing the dependencies in the files `*.dep`. Each line of this new proof consists of one of the lemmata proved before. The lines can be arranged for the manuscript using the commands described above. Then the final manuscript containing all the proofs of the lemmata can be created by the command `make_manuscript.`. It generates a file `manuscript.tex`.

Bibliography

[Da93] Dahn, B.: Applying Algebraic Properties in Deduction; preprint 1993

[DP92] Denzinger, J., Pitz, W.: Das DISCOUNT-System: Benutzerhandbuch; SEKI Working Paper SWP-92-16

[DFKBLL] U. Dunker, A. Flögel, H. Kleine Büning, J. Lehmann, Th. Lettmann: ILFA – A Project in Experimental Logic Computation, Technischer Bericht der Universität–GH Paderborn, Reihe Informatik, Nr. 142, (1994)

[BH90] Bundy, Alan; van Harmelen, Frank et all.: The Oyster-Clam System, Proc. 10th Conf. Aut. Ded., Lecture Notes Art. Int., vol. 449 (1990), 647 – 648

[LSBB92] Letz, R.; Schumann, J.; Bayerl, S.; Bibel, W.: SETHEO: A High-Performance Theorem Prover, Journal of Automated Reasoning, 8 (1992), S. 183–212

[MC90] McCune, W.: Otter 2.0, in: Stickel, M.E. (ed.): Proceedings of the 10th CADE, pp. 663-664, Springer, Berlin, 1990

[Me90] Mellouli, T.: A Tree Representation of the Modified Problem Reduction and its Extentsion to Three–valued Logic, KI–NRW 90–19. KI–Verbund NRW (1990).

[PSt81] Peterson, G. E.; Stickel, M. E.: Complete Sets of Reductions for Some Equational Theories, J. Ass. Comp. Mach. vol. 28 (1981), 231 – 264

[Pl88] Plaisted, D. A.: Non-Horn Clause Logic Programming without Contrapositives, Journal of Automated Reasoning 4 (1988), 287–325.

[Sm68] Smullyan, R. M. : First-Order Logic, Springer, Berlin 1968

[St84] Stickel, M. E.: A PROLOG Technology Theorem Prover, New generation computing 2 (1984), 371–383

[Wo92] Wolf, A. : Deduktionssysteme und Taktikübersetzung, Diplomarbeit, Humboldt-Universität zu Berlin, Fachbereich Mathematik, 1992