# LPTP: A Logic Program Theorem Prover
# (DRAFT. DO NOT COPY)

Robert F. Stärk

May 15, 2001

# Contents

# Chapter 1

# Introduction

## 1.1 What is LPTP?

LPTP is an interactive theorem prover for the formal verification of Prolog programs. It is a proof refinement system that allows a user to construct formal proofs interactively. It is possible to generate proofs *deductively* from the assumption forwards to the goal or *goal directed* from the goal backwards to the axioms. LPTP has the ability to search for proofs or parts of proofs automatically. In the simplest case, LPTP just finds the name of a lemma that already has been proved and is used at a certain point in a proof. In the best case, LPTP finds complete proofs.

LPTP has been designed for correctness proofs of pure Prolog programs. Pure Prolog programs may contain *negation*, *if-then-else* and built-in predicates like $is/2$, $</2$ and $call/n+1$. The programs, however, have to be free of *cut* and database predicates like $assert/1$ and $retract/1$ which allow to modify a program during runtime.

The kernel of LPTP is written in exactly the fragment of Prolog that can be treated in LPTP. This means that LPTP uses no single *cut*. Moreover, in principle it is possible to prove properties of LPTP in LPTP itself.

LPTP has a graphical user interface in the *Gnu Emacs Editor*. For example, the user can double-click on a quantifier and the whole scope of the quantifier is highlighted. Moreover, the Emacs LPTP-Mode provides functions that indent proofs in a correct way as this is usually done for program source code.

LPTP has a TEX- and an HTML-manager who are able to generate TEX- and HTML-output which is readable for humans.

LPTP runs under different Prologs. It has been tested with CProlog, Quintus Prolog, SICStus Prolog under ECLiPse under Unix and with Open Prolog on the Macintosh.

LPTP is able to check a 13128 lines long proof (133 pages) in 99.2 seconds for correctness (on a Sun SPARCserver 1000). Hereby it has to be said that in practise proofs or parts of proofs that have to be checked are rarely longer than

4 pages. The average response time of the system during interactive proving is therefore less than 4 seconds. In automatic theorem proving, however the response time can be arbitrary long.

A skilled user can generate up to 2000 lines of formal proofs with LPTP in one day (if the proofs are mathematically not too complicated).

LPTP uses the so-called *ground representation* for formulas and proofs.

LPTP differs from other theorem provers. At the end the user has a proof in his hands that is readable for humans and not only a message 'yes, the theorem is provable'.

We believe that in computer science the purpose of formal proofs about programs is not only to verify their correctness but also to document the code. In a similar way, in mathematics, proofs are used to communicate ideas and methods between mathematicians. A correctness proof of a computer program, for example, explains the meaning of subprocedures.

Although LPTP was a one-person project, the largest program we have verified is 635 lines long. The example program is a parser for ISO standard Prolog. The 635 lines comprise the specification of the parser, too. The specification is given by a DCG *(Definite Clause Grammar)*. The full correctness proof contains theorem like the following: if a parse tree is transformed into a token list (using *write*) and the token list is parsed back into a parse tree (using *read*), then this parse tree is identical to the original parse tree.

The fully formalized correctness proof for the ISO Prolog parser is 13000 lines long. This means that we have a factor of 20 for the full verification of this example program.

It has been often claimed that declarative programs are easier to verify than imperative programs. There exist, however, only a few examples which support this claim. There exist not many programs that have been formally verified.

One reason that it is possible to verify non-trivial programs in LPTP is the underlying logical theory. The formal system which is the basis of LPTP is a first-order predicate logic theory called *inductive extension of logic programs (IND)*. It is more or less the Clark completion of a program extended by induction.

The naive extension of Clark's theory by induction immediately leads to an inconsistency. Therefore IND is formulated in a extended language in which there are three operators **S**, **F** and **T** for *success*, *failure* and *termination*.

For a goal $G$, the expressions **S** $G$, **F** $G$ and **T** $G$ are positive formulas expressing success, failure and termination of $G$. It is an essential feature of IND that success, failure and termination are expressed by positive formulas, i.e. formulas that do not contain negation. This is the reason that induction does not lead to contradictions.

The theory IND is computationally adequate in the following sense: the formula **T** $G$ is provable in IND if, and only if, the goal $G$ is left-terminating and dynamically well-typed. The letter **T** stands for both *terminating* and *typed*.

Left-terminating means that a goal terminates independently from the selection of the clauses in the program, if always the left-most literal is selected during the computation. Dynamically well-typed means that, whenever a built-

in atom is selected, then the arguments of the built-in predicate are correctly typed such that the atom can be evaluated. Moreover, dynamically well-typed implies that negative literals are ground when they are selected.

The inductive extension of logic programs IND differs from other know theories in the following points:

1. IND is consistent for arbitrary programs.
2. IND contains induction principles. It contains simultaneous induction along the definition of predicates in a program.
3. IND contains axioms for logical built-in predicates like $is/2$, $</2$ and $call/n$. We call a built-in predicate logical, if the domain and the evaluation of the predicate are compatible with instantiation. The predicate $var/1$, for example, is not logical in this sense, since the atom $var(x)$ is evaluated to *true*, whereas the instance $var(f(0))$ has answer *no*.
4. IND is sound and complete for *arbitrary* programs and goals and not only for restricted classes of programs as for example the class of *definite* or the class of *allowed* programs.
5. IND respects the special literal selection rule of Prolog.
6. It is possible to make termination proofs in IND.
7. It is possible to treat polymorphic logic programs in IND which use the predicate $call/1$.

## On the design of this report

This report is fully accessible to the novice logic programmer and the non-proof theorist. It requires no knowledge of formal logic that goes beyond free and bound variables or substitutions. The underlying theory of LPTP however requires basic knowledge of proof-theory and inductive definitions as it is presented in the first chapters of standard textbooks (for example, in [4, 6, 7, 8, 17]). A short overview on the theoretical foundations of LPTP is given in [15]. More details can be found in [10, 11, 14, 16, 12, 13].

Although LPTP is implemented in Prolog in a declarative way, we have decided not to take the source code as description of the system. So we do not explain the implementation of LPTP in Prolog in this report, but describe the system in a neutral way. The underlying Prolog, however, will shine through at certain points. We have to assume a little familiarity with Prolog. We want to point out, however, that the purpose of the whole enterprise is to reason about pure Prolog programs in a *logical* way and not in the way a Prolog programmer usually does. Thus in principle it would be possible to use the LPTP system without ever having heard anything about Prolog.

A second point is the choice of representation of formulas. There are two possibilities, the mathematical notation or the ASCII representation. The following formula is displayed in mathematical notation:

$\forall x\,(p_1(x) \wedge p_2(x) \rightarrow q_1(x) \vee q_2(x))$.

Here is the same formula in ASCII representation:

```
all x: p1(x) & p2(x) => q1(x) \/ q2(x).
```

The mathematical notion is certainly more convenient for the eye. The user of LPTP, however, will spend most of his time at a computer in an editor and look at the ASCII representation of formulas despite the fact that LPTP is able to produce mathematical notation (TEX output) automatically. Therefore we have decided to use the ASCII represenation of formulas in this report.

When we talk about formulas, then we have to use mathematical variables like $x$, $\varphi$ and $\psi$. This means that instead of

```
all x: phi1 & phi2 => psi1 \/ psi2
```

we have to write

all $x$: $\varphi_1$ & $\varphi_2$ => $\psi_1$ \/ $\psi_2$.

Thereby we assume that $x$, $\varphi_i$ and $\psi_i$ are holes (variables) that can be filled with ASCII strings.

## 1.2 A simple example

We present the features of the LPTP system by proving some simple facts about the predicates `member` and `append`. The explanations here are extremely brief. The details are discussed later in this report.

### The 'member' program

We begin with a small Prolog program. Although Prolog has lost the '`.pl`' extension to the programming language Perl we use the traditional extension nevertheless. We invoke an editor and create the file '`list.pl`' with the following contents:

```
list([]).
list([X|L]) :- list(L).

member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).

append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

In these program clauses, identifiers that start with a capital letter like `X` and `L` are Prolog variables. In the first clause, `[]` is the empty list and, in the second clause, `[X|L]` is the same as the pair `'.'(X,L)`. The colon-dash symbol '`:-`' is

read as 'if'. The first clause for `list` says that `[]` is a list and the second clause for `list` says that the pair `[X|L]` is a list, if `L` is a list.

The predicate `member(X,L)` expresses that `X` is an element of the list `L` and the predicate `append(L1,L2,L3)` expresses that `L3` is the concatenation of `L1` and `L2`. To run the program in Prolog, we have to start Prolog and to read in the clauses with the following command (possible warnings about singleton variables can be ignored):

```
?- consult('list.pl').
```

After this is done we can ask queries. Every query has to be terminated by a full stop. Here is an example with answer 'yes':

```
?- member(2,[1,2,3]).

yes
```

The anwer 'yes' means: yes, `2` is an element of the list `[1,2,3]`. The expression `[1,2,3]` is an abbreviation for the following Prolog term:

```
'.'(1,'.'(2,'.'(3,[]))).
```

Here is an example with answer 'no':

```
?- member(7,[1,2,3]).

no
```

The answer 'no' means: no, `7` is not an element of the list `[1,2,3]`. The following query contains the Prolog variable `X`. To obtain more than one answer one has to type a semicolon.

```
?- member(X,[1,2,3]).

X = 1 ;

X = 2 ;

X = 3 ;

no
```

The first theorem below that we are going to prove about the predicate `member` will be that, whenever $\ell$ is a list, the query `member(X,`$\ell$`)` terminates and has only finitely many solutions as in this example.

## The ground representation

Since the LPTP system itself is a Prolog program it cannot deal with the Prolog code '`list.pl`' directly. LPTP needs the so-called *ground representation* of '`list.pl`' which is usually written on the file '`list.gr`'. To obtain the ground representation one has to start Prolog again and to read in the LPTP system with one of the following commands or whatever the Prolog system requires:

```
?- consult('lptp.pl').
?- load(lptp).
?- load('/home/staerk/lptp/bin/lptp').
?- [lptp].
```

Then one has to type one of the following commands:

```
?- compile_gr(list).
?- compile_gr('/home/staerk/list').
```

With the `compile_gr` command the ground representation of '`list.pl`' is written on the file '`list.gr`'. If the LPTP system does not find the file '`list.pl`' one has to type the full pathname of the file but without the '`.pl`' suffix. It is possible to define in LPTP abbreviations for path names:

```
?- set(home,'/home/staerk/').
?- compile_gr($(home)/list).
```

The syntax of pathnames and abbreviations is described in Chapter 2.

## Success, failure and termination

As we have seen in the `member` example, the basic notions of Prolog are *success*, *failure* and *termination*. Success corresponds to the answer 'yes', failure corresponds to the answer 'no' and termination has to be understood as 'only finitely many answers'.

In LPTP, there are three operators that express these properties. The operators are: `succeeds`, `fails` and `terminates`. We can ask the system about the logical meaning of the operators. To do this, we start LPTP and load the ground representation of the program clauses into the data base of LPTP with the command:

```
?- needs_gr(list).
```

Now, we can ask, for example, the following query:

```
?- def(succeeds member(?x,[?y|?m])).
```

The answer of LPTP is:

```
?x = ?y \/ succeeds member(?x,?m)
```

This formula says that `member(?x,[?y|?m])` succeeds if, and only if, `?x` is equal to `?y` or `member(?x,?m)` succeeds. The expressions `?x`, `?y` and `?m` are LPTP variables.

An LPTP variable is different from a Prolog variable. An LPTP variable (in the following called *variable*) always starts with a question mark and is followed by an identifier that begins with a lowercase letter. The symbol '`\/`' is called *disjunction* and it is read as 'or'.

In general, the success, failure or termination of a predicate is expressed by more complicated formulas. To see the exact definition of the success of the `member` predicate we type in the following query:

```
?- def(succeeds member(?x,?m)).
```

The difference to the query above is, that here the second argument of `member` is the variable `?m` and not the compound term `[?y|?m]`. The answer of LPTP is:

```
(ex l: ?m = [?x|?l]) \/
(ex [y,l]: ?m = [?y|?l] & succeeds member(?x,?l))
```

This formula says that `member(?x,?m)` succeeds, if and only, if one of the following statements is true:

1. there exists an `l` such that `?m` is equal to `[?x|?l]`, or
2. there exist `y` and `l` such that `?m` is equal to `[?y|?l]` and `member(?x,?l)` succeeds.

The symbol '`ex`' is a *quantifier* and it is read as 'there exists'. The symbol `&` is called *conjunction* and is read as 'and'.

Note that the answer to the first query can be obtained from the answer to the second query by replacing the variable `?m` everywhere by the term `[?x|?m]` and simplifying the result. This is done automatically by the system.

We can also ask the system about the definition of failure and termination. To see an example for the failure operator we type:

```
?- def(fails member(?x,[?y|?m])).
```

The answer of the system is:

```
~ ?x = ?y & fails member(?x,?m)
```

This formula says that `member(?x,[?y|?m])` fails if, and only if, it is not the case that `?x` is equal to `?y` and if `member(?x,?m)` fails. The symbol '`~`' is called *negation* and is read as 'not'. To see an example for the termination operator we type:

```
?- def(terminates member(?x,?m)).
```

The answer of the system is:

```
all [y,l]: ?m = [?y|?l] => terminates member(?x,?l)
```

This formula says that `member(?x,?m)` terminates if, and only if, for all `y` and `l`:

if `?m` is equal to `[?y|?l]`, then `member(?x,?l)` terminates.

The symbol '`all`' is called a *universal quantifier* and it is read as 'forall'. The symbol '`=>`' is called *implication* and '$\varphi$ `=>` $\psi$' is read as 'if $\varphi$ then $\psi$'.

## The first automatic proof

The first lemma that we want to prove is about the termination of the predicate `member`. We want to prove the following formula:

```
all [y,l]: succeeds list(?l) => terminates member(?y,?l)
```

Proofs are usually written on files with the extension '`.pr`'. Therefore we create in the editor a file '`example.pr`' containing the following lines:

```
:- initialize.
:- needs_gr(list).

:- lemma(member:termination,
all [y,l]: succeeds list(?l) => terminates member(?y,?l),
all [y,l]: succeeds list(?l) => terminates member(?y,?l)
 by [ind]
).
```

The command `initialize` clears the internal database of LPTP. The command `needs_gr(list)` tells LPTP to load the ground representation '`list.gr`' into its memory. The structure of the lemma is the following:

```
:- lemma(name,
 formula,
 derivation
).
```

A derivation is usually a sequence of derivation steps. At the beginning derivations are incomplete. In our case the incomplete derivation has the following form:

*formula* `by [ind]`

This tells LPTP that we want to prove *formula* by induction. We start LPTP and read in the file '`example.pr`' with one of the following commands:

```
?- consult('example.pr').
?- ['example.pr'].
?- exec('example.pr').
```

The output of the system is:

```
induction(
 [all l: succeeds list(?l) =>
    (all y: terminates member(?y,?l))],
 [step([],[],[],all y: terminates member(?y,[])),
  step([x,l],
   [all y: terminates member(?y,?l),
    succeeds list(?l)],
   [],
   all y: terminates member(?y,[?x|?l]))])
```

This is a ground Prolog term. We call it a derivation. It has the following form:

```
induction(
 [formula],
 [induction-step_1,
  induction-step_2])
```

Each induction step has the following structure:

```
step([variable,...],
 [hypothesis,...],
 derivation,
 conclusion)
```

LPTP proves the lemma by induction on the predicate `list`. There are two steps, the empty list `[]` and the compound list `[?x|?l]`. The first step is trivial, since the definition of `terminates member(?y,[])` is `tt` as can be seen by asking the system the query:

```
?- def(terminates member(?y,[])).
```

The second case is also easy, since the query

```
?- def(terminates member(?y,[?x|?l])).
```

yields the following formula, which is the hypothesis of the induction step:

```
terminates member(?y,?l).
```

Thus it is not so surprising that LPTP is able to prove the lemma automatically.

We now copy the derivation, created by the system, to the file '`example.pr`' and replace the old, incomplete, partial, gappy derivation

*formula* `by [ind]`

by the complete derivation. The file '`example.pr`' looks then as follows:

```
:- initialize.
:- needs_gr(list).

:- lemma(member:termination,
```

```
all [y,l]: succeeds list(?l) => terminates member(?y,?l),
induction(
 [all l: succeeds list(?l) =>
    (all y: terminates member(?y,?l))],
 [step([],[],[],all y: terminates member(?y,[])),
  step([x,l],
   [all y: terminates member(?y,?l),
    succeeds list(?l)],
   [],
   all y: terminates member(?y,[?x|?l]))])
).
```

The proof of the lemma is complete.

If the Emacs LPTP mode is installed correctly, then the interaction between Emacs and the LPTP process goes automatically. If we start Emacs with a file with '`.pr`' extension, then it should go into `lptp-mode`. In our example, we start Emacs with the file '`example.pr`':

```
:- initialize.
:- needs_gr(list).

:- lemma(member:termination,
all [y,l]: succeeds list(?l) => terminates member(?y,?l),
all [y,l]: succeeds list(?l) => terminates member(?y,?l)
 by [ind]
).
```

Emacs goes into `lptp-mode` and indicates it on the mode line. We can start start LPTP by selecting '`Run LPTP`' from the LPTP-menu or with the command '`M-x run-lptp`' or '`M-x run-lptp-other-frame`'. In a seperate buffer (or frame) a new Prolog process is started with LPTP. We can send the buffer '`example.pr`' to the LPTP process with the *send* command '`C-c i s`'. To replace the formula by the derivation of LPTP we set the cursor (also called *point* in Emacs) on the symbol '`by`' and copy the output of LPTP via the *get* command '`C-c i g`'. The buffer then looks as follows:

```
:- initialize.
:- needs_gr(list).

:- lemma(member:termination,
all [y,l]: succeeds list(?l) => terminates member(?y,?l),
induction(
 [all l: succeeds list(?l) =>
    (all y: terminates member(?y,?l))],
 [step([],[],[],all y: terminates member(?y,[])),
  step([x,l],
   [all y: terminates member(?y,?l),
```

```
     succeeds list(?l)],
    [],
    all y: terminates member(?y,[?x|?l]))])
).
```

Double-clicking on a keyword like `step` or `all` shows the structure of the derivation. Appendix A contains a detailed description of the Emacs LPTP mode.

## Defining an abbreviation

The predicates `list`, `member` and `append` are called *user-defined* predicates in contrast to *built-in* predicates like `is` and `call` which we will treat later. In formal proofs, there is usually a third kind of predicates which we call *abbreviations*. Here is an example. We extend the file '`example.pr`' by the following lines:

```
:- definition_pred(sub,2,
all [l1,l2]: sub(?l1,?l2) <=>
 (all x: succeeds member(?x,?l1) => succeeds member(?x,?l2))
).
```

In this predicate definition we define an abbreviation `sub(?l1,?l2)` which expresses that `?l1` is a subset of `?l2`, i.e. every element of `?l1` is also an element of `?l2`. The first thing we want to prove about `sub` is that it is transitive. To do this we add the following lines to `example.pr`:

```
:- lemma(sub:transitive,
all [l1,l2,l3]: sub(?l1,?l2) & sub(?l2,?l3) => sub(?l1,?l3),
all [l1,l2,l3]: sub(?l1,?l2) & sub(?l2,?l3) => sub(?l1,?l3)
 by [auto(9)]
).
```

An incomplete derivation like

*formula* `by [auto(9)]`

tells the system to try to prove *formula* automatically with a search depth of `9`. We send the file '`example.pr`' to LPTP and the system responds immediatly with:

```
assume(sub(?l1,?l2) & sub(?l2,?l3),
 [assume(succeeds member(?x,?l1),
   [all x: succeeds member(?x,?l1) =>
      succeeds member(?x,?l2) by elimination(sub,2),
    all x: succeeds member(?x,?l2) =>
      succeeds member(?x,?l3) by elimination(sub,2)],
   succeeds member(?x,?l3)),
  sub(?l1,?l3) by introduction(sub,2)],
 sub(?l1,?l3))
```

We take the derivation created by the system gratefully and paste it over the incomplete derivation:

*formula* `by [auto(9)]`

If we are working in Emacs then we can use the *get* command '`C-c i g`' to do this. The lemma now looks as follows:

```
:- lemma(sub:transitive,
all [l1,l2,l3]: sub(?l1,?l2) & sub(?l2,?l3) => sub(?l1,?l3),
assume(sub(?l1,?l2) & sub(?l2,?l3),
 [assume(succeeds member(?x,?l1),
   [all x: succeeds member(?x,?l1) => succeeds member(?x,?l2)
      by elimination(sub,2),
    all x: succeeds member(?x,?l2) => succeeds member(?x,?l3)
      by elimination(sub,2)],
   succeeds member(?x,?l3)),
  sub(?l1,?l3) by introduction(sub,2)],
 sub(?l1,?l3))
).
```

This proof can be read as follows: Assume that `?l1` is a subset of `?l2` and that `?l2` is a subset of `?l3`. Assume that `?x` is an element of `?l1`. By the definition of `sub` we have the following:

```
all x: succeeds member(?x,?l1) => succeeds member(?x,?l2)
all x: succeeds member(?x,?l2) => succeeds member(?x,?l3).
```

From this we obtain that `?x` is an element of `?l3`. Since `?x` was chosen arbitrarily, we have

```
all x: succeeds member(?x,?l1) => succeeds member(?x,?l3).
```

By the definition of `sub`, this means that `?l1` is a subset of `?l3`. QED.

In the derivation created by LPTP we see two implication introduction steps. In general, the derivation step `assume` has the following structure:

`assume(`*formula*$_1$`,`
*derivation*`,`
*formula*$_2$`)`

The derivations step proves the following formula:

*formula*$_1$ `=>` *formula*$_2$

*Derivation* must be a correct derivation of *formula*$_2$ from the additional assumption *formula*$_1$.

## More automatic proofs

The transitivity lemma does not use the definition of the predicate `member`. It uses the definition of the abbreviation `sub` only. The next lemma needs both:

```
:- lemma(sub:member,
all [y,l1,l2]: sub(?l1,?l2) & succeeds member(?y,?l2) =>
  sub([?y|?l1],?l2),
all [y,l1,l2]: sub(?l1,?l2) & succeeds member(?y,?l2) =>
  sub([?y|?l1],?l2) by [auto(11)]
).
```

With the increased search depth of `11` the system is able to prove this lemma with a deep thought. After we have sent the file '`example.pr`' to LPTP with the command '`C-c i s`', LPTP prints the following derivation on the screen:

```
assume(sub(?l1,?l2) & succeeds member(?y,?l2),
 [assume(succeeds member(?x,[?y|?l1]),
   [def succeeds member(?x,[?y|?l1]) by completion,
    cases(?x = ?y,
     [],
     succeeds member(?x,?l1),
     all x: succeeds member(?x,?l1) =>
       succeeds member(?x,?l2) by elimination(sub,2),
     succeeds member(?x,?l2))],
   succeeds member(?x,?l2)),
  sub([?y|?l1],?l2) by introduction(sub,2)],
 sub([?y|?l1],?l2))
```

This derivation is short enough, so we paste it over the `... by [auto(11)]` (in Emacs we simply type '`C-c i g`'). In this derivation we see for the first time a case splitting which, in general, has the following structure:

`case(`*formula*$_1$`,`
*derivation*$_1$`,`
*formula*$_2$`,`
*derivation*$_2$`,`
*formula*`)`

It expresses that *formula* is derivable in both cases, in the case of *formula*$_1$ and the case of *formula*$_2$. The following disjunction must be derivable:

*formula*$_1$ `\/` *formula*$_2$

In the derivation above, the disjunction is:

```
?x = ?y \/ succeeds member(?x,?l1)
```

It is obtained from

```
succeeds member(?x,[?y|?l1])
```

by taking its definition (which is here called completion).

Now we want to test the LPTP system to see whether it is clever. We add the following corollary at the end of the file 'example.pr':

```
:- corollary(sub:member:2,
all [x,y,i,j]: sub(?i,?j) & succeeds member(?x,?j) &
  succeeds member(?y,?j) => sub([?x,?y|?i],?j),
all [x,y,i,j]: sub(?i,?j) & succeeds member(?x,?j) &
  succeeds member(?y,?j) => sub([?x,?y|?i],?j) by [auto(5)]
).
```

As the term `corollary` suggest to any mathematician, this must follow from previously proved theorems and lemmas very easily. And indeed, LPTP finds the following derivation with thinking depth `5`:

```
assume(sub(?i,?j) & succeeds member(?x,?j) &
 succeeds member(?y,?j),
 [sub([?y|?i],?j) by lemma(sub:member),
  sub([?x,?y|?i],?j) by lemma(sub:member)],
 sub([?x,?y|?i],?j))
```

This derivation is just a two-fold application of the previous lemma.

We still have not used the predicate `append`. Our final goal for the short example session is to prove that if `?l1` is a subset of `?l` and `?l2` is a subset of `?l` then the concatenation of `?l1` and `?l2` is a subset of ?l, too. This would be too complicated for LPTP, therefore we first prove the following lemma:

```
:- lemma(append:member,
all [y,l1,l2,l3]: succeeds append(?l1,?l2,?l3) &
 succeeds member(?y,?l3) =>
  succeeds member(?y,?l1) \/ succeeds member(?y,?l2),
all [y,l1,l2,l3]: succeeds append(?l1,?l2,?l3) &
 succeeds member(?y,?l3) =>
  succeeds member(?y,?l1) \/ succeeds member(?y,?l2) by [ind]
).
```

The response of the system is the following:

```
induction(
 [all [l1,l2,l3]: succeeds append(?l1,?l2,?l3) =>
    (all y: succeeds member(?y,?l3) =>
       succeeds member(?y,?l1) \/ succeeds member(?y,?l2))],
 [step([l],
   [],
   [],
   all y: succeeds member(?y,?l) =>
     succeeds member(?y,[]) \/ succeeds member(?y,?l)),
```

```
step([x,l1,l2,l3],
 [all y: succeeds member(?y,?l3) =>
    succeeds member(?y,?l1) \/ succeeds member(?y,?l2),
  succeeds append(?l1,?l2,?l3)],
 assume(succeeds member(?y,[?x|?l3]),
  succeeds member(?y,[?x|?l1]) \/
  succeeds member(?y,?l2) by gap,
  succeeds member(?y,[?x|?l1]) \/ succeeds member(?y,?l2)),
 all y: succeeds member(?y,[?x|?l3]) =>
   succeeds member(?y,[?x|?l1]) \/
   succeeds member(?y,?l2))])
```

This derivation uses induction on the predicate `append`. Since `append` has two clauses in the file '`list.pl`', there are two steps in the induction proof. The first step, the base case, is complete. The second step is not yet complete. It contains a gap. We copy the partial derivation to the file '`example.pr`', replace the lines

```
succeeds member(?y,[?x|?l1]) \/
succeeds member(?y,?l2) by gap
```

by the following lines

```
succeeds member(?y,[?x|?l1]) \/
succeeds member(?y,?l2) by [auto(7),l(5)]
```

and send the file again to LPTP. The response of the system is:

```
[def succeeds member(?y,[?x|?l3]) by completion,
 cases(?y = ?x,
  [],
  succeeds member(?y,?l3),
  [succeeds member(?y,?l1) \/ succeeds member(?y,?l2),
   cases(succeeds member(?y,?l1),
    [],
    succeeds member(?y,?l2),
    [],
    succeeds member(?y,[?x|?l1]) \/
    succeeds member(?y,?l2))],
  succeeds member(?y,[?x|?l1]) \/
  succeeds member(?y,?l2))]
```

This is a nested case splitting. The derivation is now complete and we can step to the final theorem:

```
:- theorem(sub:append,
all [l1,l2,l3,l4]: succeeds append(?l1,?l2,?l3) &
 sub(?l1,?l4) & sub(?l2,?l4) => sub(?l3,?l4),
all [l1,l2,l3,l4]: succeeds append(?l1,?l2,?l3) &
```

```
 sub(?l1,?l4) & sub(?l2,?l4) => sub(?l3,?l4) by [auto(10)]
).
```

We send the file '`example.pr`' to the system and after a short while we get the following derivation:

```
assume(succeeds append(?l1,?l2,?l3) & sub(?l1,?l4) &
 sub(?l2,?l4),
 [assume(succeeds member(?x,?l3),
   [succeeds member(?x,?l1) \/ succeeds member(?x,?l2) by
     lemma(append:member),
    cases(succeeds member(?x,?l1),
     all x: succeeds member(?x,?l1) =>
       succeeds member(?x,?l4) by elimination(sub,2),
     succeeds member(?x,?l2),
     all x: succeeds member(?x,?l2) =>
       succeeds member(?x,?l4) by elimination(sub,2),
     succeeds member(?x,?l4))],
   succeeds member(?x,?l4)),
  sub(?l3,?l4) by introduction(sub,2)],
 sub(?l3,?l4))
```

LPTP uses the previous lemma to prove the theorem. With this proof the file '`example.pr`' is complete and we terminate the example session.

## Interactive proofs

As we have already seen, proofs can contain gaps. A gap has always the following form:

*formula* `by gap.`

It has the effect that *formula* is forced to be true at this point in the proof no matter whether it is derivable from previously proved formulas or not. The concept of gaps gives the possibility to construct proofs in arbitrary ways: from the assumptions forward to the conclusion, from the conclusion backward to the assumption or a mix of both strategies. A proof can contain an arbitrary number of gaps. We can even force the constant falsum to be true with

```
ff by gap.
```

LPTP prints a warning message on the screen for every gap it detects in a derivation. The keyword `by` is not only used for gaps. It serves for different purposes. It is used also to indicate that a formula can be derived using a lemma as in

*formula* `by lemma(append:member).`

If the expression on the right-hand side of `by` is a list as in

*formula* `by [`*tactic*`,`*option*`,...].`

then this is understood as a command for LPTP to try to prove *formula* by *tactic* and to print the result to the screen. The tactics that we have seen so far were:

- `auto(`$n$`)` — use automatic proof search of depth $n$, and
- `ind` — use induction.

There are many more tactics, for example:

- `case` — use case splitting,
- `unfold` — unfold the formula,
- `fact` — use a fact (axiom, lemma, corollary or theorem).

To show how the other tactics are used, we prove the theorem '`append:sub`' again, but this time by hand. We start with the following:

```
:- theorem(sub:append,
all [l1,l2,l3,l4]: succeeds append(?l1,?l2,?l3) &
 sub(?l1,?l4) & sub(?l2,?l4) => sub(?l3,?l4),
all [l1,l2,l3,l4]: succeeds append(?l1,?l2,?l3) &
 sub(?l1,?l4) & sub(?l2,?l4) => sub(?l3,?l4) by []
).
```

The following expression is a command for LPTP:

*formula* `by []`

The command tells LPTP to create a standard derivation for *formula*, or more precisely, to construct an initial piece of the obvious derivation for *formula* which is given by the logical structure of *formula*. In our case, the system responds with:

```
assume(succeeds append(?l1,?l2,?l3) & sub(?l1,?l4) &
 sub(?l2,?l4),
 sub(?l3,?l4) by gap,
 sub(?l3,?l4))
```

We take this proof fragment with '`C-c i g`' such that we have:

```
:- theorem(sub:append,
all [l1,l2,l3,l4]: succeeds append(?l1,?l2,?l3) &
 sub(?l1,?l4) & sub(?l2,?l4) => sub(?l3,?l4),
assume(succeeds append(?l1,?l2,?l3) & sub(?l1,?l4) &
 sub(?l2,?l4),
 sub(?l3,?l4) by gap,
 sub(?l3,?l4))
).
```

Now we change the gap to:

```
:- theorem(sub:append,
all [l1,l2,l3,l4]: succeeds append(?l1,?l2,?l3) &
 sub(?l1,?l4) & sub(?l2,?l4) => sub(?l3,?l4),
assume(succeeds append(?l1,?l2,?l3) & sub(?l1,?l4) &
 sub(?l2,?l4),
 sub(?l3,?l4) by [unfold,l(1)],
 sub(?l3,?l4))
).
```

This tells LPTP to unfold the expression `sub(?l3,?l4)` by its definition. In Emacs, we can use the command '`C-c i u`' for this. The option `l(1)` tells the system to indent its answer one columen to the right such that it fits nicely into our proof at the position we want to have it. The system responds with:

```
[assume(succeeds member(?x,?l3),
  succeeds member(?x,?l4) by gap,
  succeeds member(?x,?l4)),
 sub(?l3,?l4) by introduction(sub,2)]
```

After the command '`C-c i g`' we have:

```
:- theorem(sub:append,
all [l1,l2,l3,l4]: succeeds append(?l1,?l2,?l3) &
 sub(?l1,?l4) & sub(?l2,?l4) => sub(?l3,?l4),
assume(succeeds append(?l1,?l2,?l3) & sub(?l1,?l4) &
 sub(?l2,?l4),
 [assume(succeeds member(?x,?l3),
   succeeds member(?x,?l4) by gap,
   succeeds member(?x,?l4)),
  sub(?l3,?l4) by introduction(sub,2)],
 sub(?l3,?l4))
).
```

Now we replace the gap by the following:

```
[succeeds member(?x,?l1) \/ succeeds member(?x,?l2) by [fact],
 succeeds member(?x,?l4) by gap]
```

This tells the system that we believe that the formula

```
succeeds member(?x,?l1) \/ succeeds member(?x,?l2)
```

can be proved by a fact, i.e. lemma, corollary or theorem, but that we have forgotten the name of the fact. Indeed, LPTP finds the right lemma and answers with:

```
succeeds member(?x,?l1) \/ succeeds member(?x,?l2) by
 lemma(append:member)
```

We take the answer and have now the following incomplete proof that still contains a gap.

```
:- theorem(sub:append,
all [l1,l2,l3,l4]: succeeds append(?l1,?l2,?l3) &
 sub(?l1,?l4) & sub(?l2,?l4) => sub(?l3,?l4),
assume(succeeds append(?l1,?l2,?l3) & sub(?l1,?l4) &
 sub(?l2,?l4),
 [assume(succeeds member(?x,?l3),
   [succeeds member(?x,?l1) \/ succeeds member(?x,?l2) by
     lemma(append:member),
    succeeds member(?x,?l4) by gap],
   succeeds member(?x,?l4)),
  sub(?l3,?l4) by introduction(sub,2)],
 sub(?l3,?l4))
).
```

We replace now the expression

```
succeeds member(?x,?l4) by gap
```

by the following:

```
succeeds member(?x,?l4) by [case,l(4)]
```

In Emacs, we use the command '`C-c i c`' for this task. LPTP introduces two new gaps in its response:

```
cases(succeeds member(?x,?l1),
 succeeds member(?x,?l4) by gap,
 succeeds member(?x,?l2),
 succeeds member(?x,?l4) by gap,
 succeeds member(?x,?l4))
```

After the get command ('`C-c i g`') we have:

```
:- theorem(sub:append,
all [l1,l2,l3,l4]: succeeds append(?l1,?l2,?l3) &
 sub(?l1,?l4) & sub(?l2,?l4) => sub(?l3,?l4),
assume(succeeds append(?l1,?l2,?l3) & sub(?l1,?l4) &
 sub(?l2,?l4),
 [assume(succeeds member(?x,?l3),
   [succeeds member(?x,?l1) \/ succeeds member(?x,?l2) by
     lemma(append:member),
    cases(succeeds member(?x,?l1),
     succeeds member(?x,?l4) by gap,
     succeeds member(?x,?l2),
     succeeds member(?x,?l4) by gap,
     succeeds member(?x,?l4))],
```

```
    succeeds member(?x,?l4)),
   sub(?l3,?l4) by introduction(sub,2)],
  sub(?l3,?l4))
 ).
```

This proof now contains two gaps and our intermediate goal is to close the first one. We replace it by:

```
succeeds member(?x,?l4) by [elim,l(5)]
```

In Emacs, we use the command '`C-c i e`'. The `elim` tactic tells the system that we believe that the formula can be proved by elimination of a prior abbreviation. By default, LPTP takes the first one, in our case `sub(?l1,?l4)` and responds with:

```
[all x: succeeds member(?x,?l1) =>
   succeeds member(?x,?l4) by elimination(sub,2),
 succeeds member(?x,?l4)]
```

We take this piece of derivations with '`C-c i g`'. After that, the remaining gap is attacked in the same way as the first one. We replace it by the following:

```
succeeds member(?x,?l4) by [elim,more,l(5)]
```

In Emacs, we use '`C-u C-c i e`' for that. The option `more` tells the system to use backtracking and to search for more possibilities. In fact, LPTP first picks again `sub(?l1,?l4)` and prints the following:

```
[all x: succeeds member(?x,?l1) =>
   succeeds member(?x,?l4) by elimination(sub,2),
 succeeds member(?x,?l4) by gap]
```

This time we are asked whether we want more or not. Since the first proposal of the system still contains a gap, we type '`y`' and the system prints its second proposal which it obtains by eliminating the abbreviation `sub(?l2,?l4)`:

```
[all x: succeeds member(?x,?l2) =>
   succeeds member(?x,?l4) by elimination(sub,2),
 succeeds member(?x,?l4)]
```

This derivation is what we want and therefore we type '`n`' when we are asked for more. We type a last time '`C-c i g`' and obtain the following proof which is exactly the same proof as LPTP has found in the last section automatically:

```
:- theorem(sub:append,
all [l1,l2,l3,l4]: succeeds append(?l1,?l2,?l3) &
 sub(?l1,?l4) & sub(?l2,?l4) => sub(?l3,?l4),
assume(succeeds append(?l1,?l2,?l3) & sub(?l1,?l4) &
 sub(?l2,?l4),
 [assume(succeeds member(?x,?l3),
```

```
    [succeeds member(?x,?l1) \/ succeeds member(?x,?l2) by
      lemma(append:member),
     cases(succeeds member(?x,?l1),
      [all x: succeeds member(?x,?l1) =>
         succeeds member(?x,?l4) by elimination(sub,2),
       succeeds member(?x,?l4)],
      succeeds member(?x,?l2),
      [all x: succeeds member(?x,?l2) =>
         succeeds member(?x,?l4) by elimination(sub,2),
       succeeds member(?x,?l4)],
      succeeds member(?x,?l4))],
    succeeds member(?x,?l4)),
   sub(?l3,?l4) by introduction(sub,2)],
  sub(?l3,?l4))
).
```

## Gernerating TEX-output

The LPTP system has a TEX manager who is able to produce output which can be formatted with the program TEX. In order to generate TEX output we insert the following line at the beginning of the proof file '`example.pr`':

```
:- tex_file(example).
```

This command tells LPTP to produce TEX output and to write it on the file '`example.tex`'. At the end of the file '`example.pr`' we insert the following command:

```
:- bye.
```

This command tells LPTP to close all files, especially the file '`example.tex`'. We can format the file '`example.tex`' with `tex` and view the result in a previewer like for example `xdvi`.

```
unix> tex example.tex
unix> xdvi example.dvi
```

The file '`example.dvi`' can be translated into a PostScript file by `dvips`.

## Generating HTML-output

LPTP has an HTML-manager, too. This manager is not built into the system but it constists of two external Perl scripts '`pl2html.perl`' and '`pr2html.perl`'. The first script converts Prolog files with extension '`.pl`' into HTML, whereas the second script converts proof files with extension '`.pr`' into HTML. In our example the scripts are used as follows:

```
unix> mkdir html
unix> pl2html.perl -o html list.pl
unix> pr2html.perl -o html example.pr
```

These commands create several HTML-files. The main file is '`example.html`'. It can be viewed, for example, with the following command:

```
unix> netscape html/example.html
```

The file '`example.html`' contains a list of all definitions, lemmas, corollaries and theorems of '`example.pr`'. Each item has a link to the corresponding file.

*In the Middle Ages, knowledge of Latin and Greek was essential for all scholars. The one-language scholar was necessarily a handicapped scholar who lacked the perception that comes from seeing the world from two points of view. Similarly, today's practitioner of Artificial Intelligence is handicapped unless thoroughly familiar with both Lisp and Prolog, for knowledge of the two principal languages of Artificial Intelligence is essential for a broad point of view.*

*P. H. Winston, 1986, in*
*'Prolog Programming for Artificial Intelligence' by I. Brakto.*

# Chapter 2

# Basic concepts of LPTP

## 2.1 The structure of proof files

Proof files have a *header* and a *main* part. To Prolog programmers, proof files look like Prolog code files and in fact they are. Proof files consist of *commands* for the LPTP system. Proof files are written in an editor in the same way as programs are written. The difference is that proof files are checked for correctness far more often than programs are compiled. We recommend to write proof files in the Emacs editor with a special Emacs mode that is described in Appendix A.

### The header of a proof file

The header of a proof file contains a command for initialization and commands for input and output. These commands are similar to import and export statements in programming languages with modules.

```
:- initialize.

:- thm_file(path).   % Write theorems to path.thm
:- tex_file(path).   % Write TeX output to path.tex

:- needs_gr(path).   % Read in clauses from path.gr
:- needs_thm(path).  % Read in the theorem file path.thm
```

The exact meaning of the commands and the syntax of *path* is explained below.

### The main part of a proof file

The main part of a proof file contains lemmas, theorems and corollaries. It contains also definitions for predicate and function symbols and axioms for built-in predicates. The main part of a proof file corresponds in programming languages to the part of a file where procedures are defined.

Lemmas, theorems and corollaries contain a *reference*, a *formula* and a *derivation*. The *reference* is a name by which we can refer to the lemma, theorem or corollary. It must be a colon separated list of names like, for example:

```
plus:associative
```

The notions of *formula* and *derivation* are explained in Chapter 3. For the moment it is best to think of a *derivation* as a body of a procedure. Lemmas have the following form:

```
:- lemma(reference,
 formula,
 derivation
).
```

Theorems have the following form:

```
:- theorem(reference,
 formula,
 derivation
).
```

Corollaries have the following form:

```
:- corollary(reference,
 formula,
 derivation
).
```

Predicate definitions are used to define abbreviations. They are described in detail in Section 4.1. Predicate definitions have the following form:

```
:- definition_pred(reference,
 formula
).
```

Functions symbols are used to extend the language. They are described in Section 4.2. Function definitions have the following form:

```
:- definition_fun(reference,
 formula,
 existence by reference,
 uniqueness by reference
).
```

Axioms are used mainly for built-in predicates. Axioms are described in more detail in Section 4.3. They have the following form:

```
:- axiom(reference,
 formula
).
```

Proof files are terminated with the `bye` command.

```
:- bye(file).
```

### Comments

Any text between `/*` and `*/` is considered as comment. One line comments start with the `%` character and end with the next line terminator. Comments are ignored by LPTP. The program `pr2html.perl` treats comments starting with `/**` in a special way.

## 2.2 Checking proof files

How does the LPTP system work? — Proof files are written in an editor and then executed by the LPTP system. The execution of a proof file is similar to compilation of a program. First, a proof file is parsed and transformed to an internal representation. The parsing is done by the Prolog interpreter, since proof files have Prolog syntax. The transformation into the internal representation is done by LPTP.

In a next step, the derivations are checked for correctness. If a derivation contains a gap with a tactics command, the LPTP system tries to apply the tactics. For example, the LPTP system tries to extend a derivation at the specific point indicated by the tactics command. If the proof is correct, the theorem is added to the internal database such that it can be used later. The internal represenation of the theorem is also written to the so-called theorem file such that it can be used later in other proof files.

### The internal database

The LPTP system has an internal database that contains

- lemmas, theorems, corollaries,
- predicate definitions, function definitions,
- axioms for built-in predicates,
- program clauses.

The exact representation of the internal database is not important. It may help, however, to understand the system by assuming that the internal database is a collection of facts of the following form:

- `db__lemma(`*reference*`,`*formula*`).`
- `db__theorem(`*reference*`,`*formula*`).`
- `db__corollary(`*reference*`,`*formula*`).`
- `db__pred(`*predicate_name*`,`*arity*`,`*formula*`).`
- `db__fun(`*function_name*`,`*arity*`,`*formula*`).`
- `db__axiom(`*reference*`,`*formula*`).`

- `db__clauses(`*predicate_name*`,`*arity*`,`*clause_list*`).`

For Prolog programmers, it is important to note that the program clauses in the internal database of LPTP have nothing to do with the internal representation of Prolog clauses of the underlying Prolog interpreter. LPTP is implemented in Prolog but has its own internal database.

## Commands

The LPTP system has two kinds of commands: (i) commands that are used in proof files and (ii) commands that are used interactively in the LPTP interpreter. We indicate the category of a command by using the symbol '`:-`' for proof file commands and '`?-`' for LPTP commands. This means that commands of the kind (i) are written as follows:

`:-` *command*`(`*argument*`,...).`

Commands of the kind (ii) are written as follows:

`?-` *command*`(`*argument*`,...).`

Note, that all commands are terminated by a period.

## Commands in a proof file

The proof file commands have the following effects on the internal database:

`:- initialize.`
: This command should always be the first command in a proof file. It clears the internal database completely.

`:- needs_thm(`*path*`).`
: This command reads in the the file *path*`.thm` and adds all the lemmas, theorems, corollaries, definitions and axioms from *path*`.thm` to the internal database. Any number of `needs_thm` commands can occur in a proof file.

`:- needs_gr(`*path*`).`
: This command reads in the file *path*`.gr` and adds all the program clauses from *path*`.gr` to the internal database. Any number of `needs_gr` commands can occur in a proof file.

`:- thm_file(`*path*`).`
: This command tells LPTP to write theorem output that can be used later in other proofs using the `needs_thm` command. The `thm_file` command opens the file *path*`.thm` for writing. During proof-checking the internal representation of lemmas, theorems, corollaries, definitions and axioms are written to this file.

`:- tex_file(`*path*`).`
: This command tells LPTP to write TEX output. The command opens the file *path*`.tex` and writes TEX code to *path*`.tex` during proof checking. The TEX code can be later used by TEX or LATEX to create formatted output.

`:- prt_file(`*path*`).`
: This command tells LPTP to write pretty print output. The command opens the file *path*`.prt` and writes correctly indented ascii output to the file. This command is not very often used.

`:- lemma(`*reference*`,`*formula*`,`*derivation*`).`
: This command is one of the most used commands of the LPTP system. It checks whether *derivation* is a correct derivation for *formula* (see p. 50). If this is the case then *formula* is added to the internal database as a lemma. It can be referred to in subsequent proofs by *reference*. The *formula* together with the *reference* is also written to the '`.thm`' file such that it can be used later in other proof files. The *reference* has to be a colon separated list of names. If *derivation* contains tactic commands then LPTP performs proof search to complete the tactics and writes the sub-derivations to the standard output.

`:- theorem(`*reference*`,`*formula*`,`*derivation*`).`
: This command works in the same way as the `lemma` command.

`:- corollary(`*reference*`,`*formula*`,`*derivation*`).`
: This command works in the same way as the `lemma` command.

`:- definition_pred(`*reference*`,`*formula*`).`
: This command checks whether *formula* is a correct definition for a defined predicate symbol also called an abbreviation. If this is the case then the definition is added to the internal database and the abbreviation can be used in subsequent derivations. The definition is also written to the '`.thm`' file. For more details, see Section 4.1.

`:- definition_fun(`*ref*`,`*formula*`,existence by `*ref*`,uniqueness by `*ref*`).`
: This command checks whether *formula* is a correct definition of a function symbol. If this is the case then the definition is written to the internal database and the defined function symbol can be used in subsequent derivations. The definition is also written to the '`.thm`' file. For more details, see Section 4.2.

`:- axiom(`*reference*`,`*formula*`).`
: This command works as the *lemma* command except that there is no *derivation* argument that has to be checked. Axioms should only be used in the context of built-in predicates and only in accordance to the criteria of [16]. For more details, see Section 4.3.

`:- bye(`*file*`).`
: This command should always be the last command in a proof file. It closes all files that have been opened for writing.

### File name extensions

The LPTP system relies on certain file name extensions. Therefore, the following file name extensions have to be used:

*file*`.pl`
: Prolog source code files; contain the source code of the the programs that are to be verified. Contain also predicates that are used in specifications.

*file*`.pr`
: Proof files; contain formal proofs of properties of the programs in the '`.pl`' files.

*file*`.gr`
: Ground representation files; contain the ground representation of programs that are to be verified. These files are created from the '`.pl`' files with the `compile_gr(`*file*`)` command.

*file*`.thm`
: Theorem files; contain the internal representation of theorems, lemmas, corollaries and axioms. Contain also the definition of defined predicate and function symbols. Generated by the `thm_file(`*path*`)` command at the beginning of proof files.

*file*`.tex`
: TEX file; contain TEX code created from '`.pr`' files. Generated by the `tex_file(`*path*`)` command at the beginning of proof files.

All but the Prolog source code and the proof files are automatically generated files that can safely be deleted without loosing any information. The ground representation files can be generated from the source code files. The theorem- and TEX files can be generated from the proof files.

### Syntax of path names

The LPTP system has an archaic way to refer to files. A *path* has always to be the full path name to a file. — This is not as cumbersome as it appears, since you can define aliases for paths in LPTP similar to shell languages.

LPTP uses the following conventions: The path separator of LPTP is the slash '`/`'. An alias is used in the form `$(`*alias*`)`. A suffix to a file name is inserted automatically. A typical command in LPTP looks as follows:

```
:- needs_thm($(lib)/list/permutation).
```

This command reads in the file `permutation.thm` which is in the subdirectory `list` of the LPTP library. The `$(lib)` is expanded into the full path name to the library, since `lib` is a predefined alias. Thus the above path name may expand to

```
/home/staerk/lptp/list/permutation.thm
```

on a Unix system and to

```
:HD:lptp:list:permutation.thm
```

on a Macintosh computer. Note that LPTP automatically converts the slashes to colons on the Macintosh.

The following commands can be used to define an alias and to expand it (in case of problems):

`:- set(`*alias*`,`*path*`).`
: This commands defines *alias* as an abbreviation for *path*. It can later be used in path names in the form `$(`*alias*`)`.
Example: `?- set(tmp,'/home/staerk/tmp').`

`?- io__expand(`*path*`,X).`
: This command is used to expand *path* into its full form in case of problems with file names. Note, that there are two underscores in this command. Actually, this is an internal predicate of LPTP.

`$(lptp)`
: This is a predefined alias that refers to LPTP's home directory.

`$(lib)`
: This is a predefined alias that refers to the library of LPTP. It is defined as `$(lptp)/lib`. See Chapter 6 for a survey of the library.

`$(examples)`
: This is a predefined alias that refers to the `examples` subdirectory of LPTP. It is defined as `$(lptp)/examples`. See Chapter 7 for a description of the examples.

`$(tex)`
: This is a predefined alias that refers to the `tex` subdirectory of LPTP. It is defined as `$(lptp)/tex`.

`$(tmp)`
: This is a predefined alias that refers to the `tmp` subdirectory of LPTP. It is defined as `$(lptp)/tmp`.

Maybe it is easier to look at an example. A typical header in a proof file for the verification of the quicksort program may look as follows:

```
:- set(home,'/home/staerk').
:- set(tmp,$(home)/tmp).
:- set(exp,$(home)/examples).

:- needs_thm($(lib)/list/list).
:- needs_thm($(lib)/list/permutation).
:- needs_gr($(exp)/quicksort).
:- thm_file($(exp)/quicksort).
:- tex_file($(tmp)/quicksort).
```

First an alias `home` is defined. The `home` alias is then used to define an alias `tmp` for a temporary directory that contains junk files and the alias `exp` for a directory that contains the quicksort program that has to be verified. The `lib` alias is predefined and refers to the LPTP library.

If all else fails then one can use full path names between single quotes like, for example:

```
:- needs_thm('/home/staerk/lptp/list/permutation').
```

For Prolog programmers it has to be mentioned that a path is neither a string nor an atom but it is structured ground Prolog term that is expanded by LPTP into a file name. Thus the slash `/` is a binary infix constructor and the dollar `$` is a unary constructor symbol.

It is a bad idea to use underscores or spaces in file names, since later TEX or other programs may have difficulties. Note also that in the above example the alias `home` has nothing to do with the shell environment variable `HOME`. The string `home` is a Prolog atom whereas `HOME` is a Prolog variable. The command `set(HOME,...)` is meaningless in LPTP.

# Chapter 3

# Formal proofs

In this chapter we introduce the notion of a formal proof. We have ordered the inference rules and axioms according to statistical data from 47000 lines of formal proofs. This means that the most frequently used inference rules and axioms come first and the rarely used inference rules and axioms come last. However, the statistical data reflect only the personal style of formal proofs of the author. The reader is completely free to use the inference rules and axioms in a different manner. The general style of proofs, namely natural deduction style, is built-in to the LPTP system and cannot be changed by the user.

## 3.1 Syntax and grammar

LPTP uses the syntax of standard Prolog and LPTP files are parsed by the built-in parser of the Prolog interpreter. Therefore we describe below a subset of the syntax of standard Prolog which comprises the syntax of formulas and derivations.

To Prolog programmers it may be of help if we point out that all LPTP objects are ground Prolog expressions. The syntactic categories *variable*, *term*, *atom*, *goal*, *formula* and *derivation* below refer to objects of LPTP and not to constructs of Prolog. For example, by a *variable* we mean a ground Prolog term `?x` and not a Prolog variable `X`.

To logicians it may be useful if we mention that ground Prolog terms are nothing else then finite trees. We encode terms, formulas and derivations as finite trees.

We refer the reader interested in details of Prolog syntax, especially in prefix, infix and postfix operators, to the example in Section 7.3. There we present a parser for ISO standard Prolog which has been fully verified with LPTP.

### Tokens

The input stream is divided into *tokens*. Tokens are names or the separator symbols. There can be any amount of layout characters between tokens. Separator symbols are:

```
( ) , [ ] |
```

Round brackets are used for grouping, whereas square brackets are used to construct lists. The comma and the vertical bar are used to separate arguments in compound terms and to separate elements of a list. There are three kinds of names: identifier names, graphic names and single-quoted names. Between two names of the same kind there has to be at least one layout character.

### Identifier names

An identifier name is a sequence of alpha numeric characters and the underscore that starts with a lower case letter. Examples of identifier names are:

```
x1   l1   member   quick_sort
```

LPTP treats integers, i.e. sequences of digits, as names, too.

### Graphic names

A graphic name is a sequence of graphic characters. Graphic characters are:

```
# $ & * + - . / : < = > ? @ \ ^ ~
```

Examples of graphic names are:

```
--   <=>   @+
```

A graphic name should not start with `/*` because this is the beginning of a comment. The graphic name '`.`' (full stop) followed by layout space has a special meaning to Prolog. It is a terminator token.

### Single-quoted names

A single quoted name is any sequence of character enclosed in single quotes that does not contain single quotes. Example of single quote names are:

```
'test.pl'   '.'   'TeX'
```

The single quotes are not part of the name. Thus the name `'abc'` is the same as `abc`.

## References

References are used to identify lemmas, theorems and corollaries. A *reference* is a colon separated list of names:

*name*:*name*: ... :*name*

As a special case a single *name* is also considered as a reference.

## Notation

In the following with use words in italic like *term*, *atom*, *goal*, *formula* and *derivation* as non-terminals in the sense of a formal grammar. Terminals are written in type-writer font like, for example, `=>` or `all`. When we talk about the elements of a syntactic category defined by a non-terminal we use the following meta-symbols:

$x$, $y$, $z$, ..... for variables,

$s$, $t$, ..... for terms,

$A$, $B$, ..... for atoms,

$G$, $H$, ..... for goals,

$\varphi$, $\psi$, $\chi$, ..... for formulas,

$\Gamma$, $\Delta$, $\Pi$, ..... for finite sequences of formulas,

$d$, $e$, ..... for derivations.

## Terms

Terms are the syntactic concepts that denote data. Terms should be considered as finite trees. A Prolog program defines relations between terms. A term is either a variable, a constant or a compound term. Here is the grammar for terms:

*term* → `?`*name*
| *name*
| *name*`(`*term*`,...,`*term*`)`
| `[]`
| `[`*term*`,...,`*term*`]`
| `[`*term*`,...,`*term*`|`*term*`]`
| *term op term*
| *op term*
| *term op*

The single productions of the grammar have the following meaning:

`?`*name*

Terms of this kind are called *variables*. Note, that variable names start with a lower case letter or are integers. The question mark indicates that they are variables and not constants. Examples of variables are:
`?x`, `?l1`, `?3`.

*name*

Terms of this kind are called *constants*. The constant `x` is not the same as the variable `?x`. Examples of constants are:
`c`, `open_bracket`.

*name*`(`*term*`,...,`*term*`)`

Terms of this kind are called *compound terms*. Sometimes they are also called *constructor terms*. *Name* is called a *function symbol* or a *constructor*. The *terms* are called its *arguments*. The number of arguments is called the *arity* of the constructor. Examples of compound terms are:
`pair(?x,?y)`, `f(?u,c,g(?v))`, `++(?z)`.

`[]`

This term is called the *empty list*. It is a special constant. Sometimes it is also called *nil*. However, the constant `nil` is not the same as `[]`.

`[`*term*`,...,`*term*`]`

Terms of this kind are called *lists*. A list `[`$t_1$`,...,`$t_n$`]` is an abbreviation for the compound term `'.'(`$t_1$`,...,'.'(`$t_n$`,[]))`. Thus the list `[1,2]` is exactly the same as the term `'.'(1,'.'(2,[]))`. Examples of lists are:
`[1,2,3]`, `[?x,?y]`, `[f(f(?z))]`.

`[`*term*`,...,`*term*`|`*term*`]`

Terms of this kind are also called *lists*. The difference to the lists above is that they are not nil-terminated. A list `[`$t_1$`,...,`$t_n$`|`$t_{n+1}$`]` is an abbreviation for the compound term `',(`$t_1$`,...,'.'(`$t_n$`,`$t_{n+1}$`))`. Thus the list `[1,2|3]` is exactly the same as the term `'.'(1,'.'(2,3))`. Examples of lists are:
`[?head|?tail]`, `[?x,?y|?z]`.

*term op term*

Terms of this kind are called *infix terms*. The name *op* must be declared in Prolog as an infix operator with a certain precedence and associativity. An infix term $t_1$ *op* $t_2$ just a convenient abbreviation for the compound term *op*`(`$t_1$`,`$t_2$`)`. Thus the term `1 + 2` is exactly the same as the term `+(1,2)`. Examples of infix terms are:
`?x + ?y * ?z`, `?n + 1`.

*op term*

Terms of this kind are called *prefix terms*. The name *op* must be declared in Prolog as a prefix operator with a certain precedence. A prefix term *op* $t$ is just a convenient abbreviation for the compound term *op*`(`$t$`)`. Thus the prefix term `- ?x` is exactly the same as the compound term `-(?x)`.

*term* *op*
: Terms of this kind are called *postfix terms*. The name *op* must be declared in Prolog as a postfix operator with a certain precedence. A postfix term *t* *op* is just a convenient abbreviation for the compound term *op*`(`*t*`)`. Thus the postfix term `?x ++` is exactly the same as the term `++(?x)`.

Parentheses are used to group terms with operators. The usual precedence rules of algebra apply. For example, the term `?m + ?i * ?j` is the same as the term `?m + (?i * ?j)`.

## Goals

Goals have three different purposes. First, they are the same as the bodies of program clauses. Second, they are used to query a Prolog program. Finally, in our context, they are used to form atomic formulas of the form `succeeds` *G*, `fails` *G* and `terminates` *G*. Here is the grammar for goals:

*goal* → `true`
| `fail`
| *term* `=` *term*
| *atom*
| `~`*goal*
| *goal* `&` `...` `&` *goal*
| *goal* `\/` `...` `\/` *goal*

*atom* → *name*
| *name*`(`*term*`,...,`*term*`)`

The single productions of the grammar have the following meaning:

`true`
: The goal that always succeeds.

`fail`
: The goal that always fails.

*term* `=` *term*
: Goals of this kind are called *equations*. Equations are solved by unification. Examples of equations are:
`?l3 = [?x|?l1]`, `f(?x,?y) = f(1,2)`.

*name*
: Goals of this kind are called *propositional atoms*. *Name* is called a predicate symbol of arity zero. Examples of propositional atoms are:
`loop`, `start`.

*name*`(`*term*`,...,`*term*`)`
: Goals of this kind are called *atomic goals*. Atomic goals define relations between terms. *Name* is called a*predicate symbol*. The *terms* are called its *arguments*. The number of arguments is called the *arity* of the predicate. Example of atomic goals are:
`member(?x,[?y|?l])`, `append(?l1,?l2,?l3)`.

`~`*goal*
: Goals of this kind are called *negated goals*. The meaning of the negation symbol is *negation by finite failure*. If the attempt to compute $G$ fails after finitely many steps then `~`$G$ succeeds.

*goal* `&` `...` `&` *goal*
: Goals of this kind are called conjunctions. Conjunctions are sequential conjunctions. This means that to solve $G_1$ `&` $G_2$ one has to solve first $G_1$ and then $G_2$.

*goal* `\/` `...` `\/` *goal*
: Goals of this kind are called disjunctions. Disjunctions are alternatives. To solve a disjunction $G_1$ `\/` $G_2$ first the goal $G_1$ is tried. If it succeeds then the disjunction succeeds. If it fails then $G_2$ is tried and the disjunction succeeds or fails according to whether $G_2$ succeeds or fails.

The description of goals is only informal. Their exact declarative meaning is given by the axioms and rules for the operators `succeed`, `fail` and `terminates` in Section 3.3. The operational meaning is given be the model of pure Prolog in [16].

## Formulas

Formulas are mathematical statements. The simplest formulas are equations and logical relations. Formulas can be true or false. Derivations are used to show that formulas are valid or follow from certain assumptions. Here is the grammar for formulas:

*formula* → `tt`
| `ff`
| *term* `=` `...` `=` *term*
| *term* `<>` *term*
| `gr(`*term*`)`
| *name*
| *name*`(`*term*`,...,`*term*`)`
| `succeeds` *goal*
| `fails` *goal*
| `terminates` *goal*
| `def succeeds` *atom*
| `def fails` *atom*

| `def terminates` *atom*
| `~`*formula*
| *formula* `&` `...` `&` *formula*
| *formula* `\/` `...` `\/` *formula*
| *formula* `=>` *formula*
| *formula* `<=>` *formula*
| `all` *name*`:`*formula*
| `all [`*name*`,...,`*name*`]:`*formula*
| `ex` *name*`:`*formula*
| `ex [`*name*`,...,`*name*`]:`*formula*

The single productions of the grammar have the following meaning:

`tt`
: The constant *true*, also called *verum*. The constant `tt` is considered as the same as the empty conjunction.

`ff`
: The constant *false*, also called *contradiction* or *falsum*. The constant `ff` is considered as the same as the empty disjunction.

*term* `=` `...=` *term*
: Formulas of this kind are called *equations*. The meaning of a chain of equations $t_1$ `=` $t_2$ `=` $t_3$ is that $t_1$ is equal to $t_2$ and $t_2$ is equal to $t_3$. Examples of equations are:
`?x=?y`, `[?x|?l1] ** ?l2 = [?x|?l1 ** ?l2] = [?x|?l3]`.

*term* `<>` *term*
: The meaning of the formula $t_1$ `<>` $t_2$ is that $t_1$ is not equal to $t_2$, in other words that $t_1$ is different from $t_2$. Examples are:
`[] <> [?x|?l]`, `0 <> s(?x)`.

`gr(`*term*`)`
: The meaning of `gr(`$t$`)` is that the term $t$ is ground, i.e. does not contain variables during runtime. Examples are:
`gr(c)`, `gr([?x|?l])`.

*name*
: Formulas of this kind are called *propositional constants* or *nullary predicate symbols*.

*name*`(`*term*`,...,`*term*`)`
: Formulas of this kind are called *atomic formulas*. *Name* is called a *predicate symbol*. It expresses a relation between its arguments. The number of arguments is the *arity* of the predicate symbol. Examples of atomic formulas are:
`sub(?l1,?l2)`, `err_msg(?x,?l)`.

`succeeds` *goal*
: The formula `succeeds` $G$ expresses that the goal $G$ succeeds. Examples for formulas of this kind are:
`succeeds member(?x,[?x|?l])`, `succeeds append([1,2],[3,4],?l)`.

`fails` *goal*
: The formula `fails` $G$ expresses that the goal $G$ fails in the sense of *finite failure*. Examples for formulas of this kind are:
`fails member(?x,?l)`, `fails append([?x|?l1],?l2,?l3)`.

`terminates` *goal*
: The formula `terminates` $G$ expresses that the goal $G$ terminates and is dynamically well-typed. Termination means universal termination. The whole computation tree for $G$ must be finite. Well-typed means that there are no errors in any branch of the computation. All negated goals are ground if they are called and all built-in atoms are instantiated correctly. Examples for formulas of this kind are:
`terminates (member(?x,?l1) & ~member(?x,?l2))`.

`def succeeds` *atom*
: The formula `def succeeds` $A$ is an abbreviation of a formula that is obtained from the program clauses for the atom $A$ and expresses the success of $A$. The exact definition is known to the LPTP system and can be seen by asking the following command:
`?- def(succeeds` $A$`).`

`def fails` *atom*
: The formula `def fails` $A$ is an abbreviation of a formula that is obtained from the program clauses for the atom $A$ and expresses the finite failure of $A$. The exact definition is known to the LPTP system and can be seen by asking the following command:
`?- def(fails` $A$`).`

`def terminates` *atom*
: The formula `def terminates` $A$ is an abbreviation of a formula that is obtained from the program clauses for the atom $A$ and expresses the termination of $A$. The exact definition is known to the LPTP system and can be seen by asking the following command:
`?- def(terminates` $A$`).`

`~`*formula*
: This kind of formula is called *negation*. The connective `~` is read as *not*. The formula `~`$\varphi$ expresses that $\varphi$ is not true.

*formula* `&` `...` `&` *formula*
: This kind of formula is called *conjunction*. The connective `&` is read as *and*. The formula $\varphi_1$ `&` `...` `&` $\varphi_n$ expresses that all formulas $\varphi_1, \ldots, \varphi_n$ are true.

*formula* `\/ ... \/` *formula*

This kind of formula is called *disjunction*. The connective `\/` is read as *or*. The formula $\varphi_1$ `\/ ... \/` $\varphi_n$ expresses that at least one of the formulas $\varphi_i$ is true.

*formula* `=>` *formula*

This kind of formula is called *implication*. The connective `=>` is read as *implies* or *if then*. The formula $\varphi$ `=>` $\psi$ expresses that, if $\varphi$ is true, then $\psi$ is true.

*formula* `<=>` *formula*

This formula is called *equivalence*. The connective `<=>` is read as *is equivalent to*. The formula $\varphi$ `<=>` $\psi$ expresses that $\varphi$ is true if, and only if, $\psi$ is true.

`all` *name*`:`*formula*

The symbol `all` is called *universal quantifier*. The formula `all` $x$`:`$\varphi$ expresses that, for all $x$, the formula $\varphi$ is true. The quantifier `all` $x$ is read as *forall* $x$. The formula $\varphi$ is said to be *in the scope* of the universal quantifier. The quantifier *binds* the variable $x$ in $\varphi$. All occurences of the variable $x$ in $\varphi$ are called *bound* occurences. An example of a universaly quantified formula is:

```
all x:r(?x).
```

Note, that there is no question mark before the first `x`. It is a mistake to write `all ?x:r(?x)`.

`all [`*name*`,...,`*name*`]:`*formula*

It is possible to quantify over more than one variable. The formula `all [`$x_1$`,...,`$x_n$`]:`$\varphi$ expresses that, for all $x_1, \ldots, x_n$, the formula $\varphi$ is true. The variables $x_1, \ldots, x_n$ are bound in $\varphi$. The LPTP system identifies the formula `all x:(all y:`$\varphi$`)` with `all [x,y]:`$\varphi$. The formula `all [x]:`$\varphi$ is the same as `all x:`$\varphi$. An example of a universally quantified formula is:

```
all [x,l]:succeeds list(?l) => terminates member(?x,?l).
```

Note that the scope of the quantifier extends as far to the right as possible.

`ex` *name*`:`*formula*

The symbol `ex` is called *existential quantifier*. The formula `ex` $x$`:`$\varphi$ expresses that, there exists an $x$ such that the formula $\varphi$ is true. The quantifier `ex` $x$ is read as *there exists an* $x$ or *there is an* $x$. The formula $\varphi$ is said to be *in the scope* of the existential quantifier. The quantifier *binds* the variable $x$ in $\varphi$. All occurences of the variable $x$ in $\varphi$ are called *bound* occurences. Note how one has to use parentheses in the following example:

```
all l:succeeds list(?l) => (ex n:succeeds lenght(?l,?n)).
```

`ex [`*name*`,...,`*name*`]:`*formula*

It is possible to quantify over more than one variable. The formula `ex [`$x$`,`$y$`]:`$\varphi$ expresses that there exist $x$ and $y$ such that $\varphi$ is true.

We write $\varphi \equiv \psi$ if the formula $\varphi$ is syntactically equal to $\psi$ modulo alpha conversion (renaming of bound variables). We write $s \equiv t$ it the term $s$ is syntactically equal to $t$. By $\mathrm{FV}(\varphi)$ we denote the set of free variables of $\varphi$. A variable is free in $\varphi$ if it is not bound by a quantifier. For example, the variable `y` is free in the fomula `all x:r(?x,?y)` but the variable `x` is not.

## Precedences and parentheses

The following table expresses the associativity and precedence of the operators that are used in formulas and terms. (The operator `by` will by introduced later.)

```
:- op(980,xfy,by).
:- op(970,xfy,:).
:- op(960,yfx,<=>).
:- op(950,xfy,=>).
:- op(940,yfx,\/).
:- op(930,yfx,&).
:- op(900,fy,~).
:- op(900,fy,not).
:- op(900,fy,def).
:- op(900,fy,succeeds).
:- op(900,fy,fails).
:- op(900,fy,terminates).
:- op(800,fy,all).
:- op(800,fy,ex).
:- op(700,yfx,=).
:- op(700,xfy,<>).
:- op(100,fy,?).
```

The higher the precedence of an operator the lower is its binding strength. The operator with the highest precedence in an expression is the main operator of an expression. For example, the formula

$\varphi_1$ `&` $\varphi_2$ `=>` $\psi_1$ `\/` $\psi_2$

is the same as the formula

`(`$\varphi_1$ `&` $\varphi_2$`) => (`$\psi_1$ `\/` $\psi_2$`).`

The tag `xfy` expresses that the operator is an infix operator that associates to the right. For example, the formula

$\varphi$ `=>` $\psi$ `=>` $\chi$

is the same as the formula

$\varphi$ `=> (`$\psi$ `=>` $\chi$`).`

The tag `yfx` expresses that the operator is an infix operator that associates to the left. For example, the formulas

$\varphi$ `&` $\psi$ `&` $\chi$, $\varphi$ `\/` $\psi$ `\/` $\chi$

are the same as the formulas

`(`$\varphi$ `&` $\psi$`)` `&` $\chi$, `(`$\varphi$ `\/` $\psi$`)` `\/` $\chi$.

Note, that the colon operator has a high priority such that that formula

```
ex x:r(?x) & q(?x)
```

means

```
ex x:(r(?x) & q(?x))
```

and not

```
(ex x:r(?x)) & q(?x).
```

## Internal representation of formulas

It may help to understand the system if we explain how terms and formulas are represented internally. Although terms and formulas are themselves Prolog expressions the LPTP system translates them into a uniform representation. The uniform representation only knows variables, compound expressions and abstractions as given by the following grammar:

*expression* → `$(`*name*`)`
| `[`*tag*`,`*expression*`,...,`*expression*`]`
| `@(`*tag*`,[`*name*`,...,`*name*`],`*expression*`)`

For example, the formula

```
all [x,y]:r(?x,f(?y),c)
```

is translated into the internal expression

```
@(forall,[x,y],[r/2,$(x),[f/1,$(y)],[c/0]]).
```

The uniform representation of expressions has the advantage that the internal predicates of LPTP for substitution and matching always have only three cases. It gives also the possibility to add new syntactic constructs to the system without the nightmare of changing a lot of predicates.

## Automatic flattening of formulas

It is during the translation into uniform expressions that conjunctions, disjunctions and quantifiers are automatically flattened. For example, a conjunction

`(`$\varphi_1$ `&` $\varphi_2$`)` `&` `(`$\varphi_3$ `&` `tt)`

is automatically translated into the internal representation

`[&,`$\bar{\varphi}_1$`,`$\bar{\varphi}_2$`,`$\bar{\varphi}_3$`].`

Nested quantifiers of the same type are also collected into one abstraction. For example, the formula

`ex x:(ex y:(ex z:`$\varphi$`))`

is translated into

`@(exists,[x,y,z],`$\bar{\varphi}$`).`

The inference rules below rely on the fact that the flattening is done.

## Transformation of goals into formulas

The LPTP computes for each goal $G$ three formulas $S(G)$, $F(G)$ and $T(G)$ which express success, failure and termination of $G$. The user never has to call the functions $S$, $F$, and $T$, because they are built into the system. However, it is good to know how they are defined. The idea is that the operators `succeeds`, `fails` and `terminates` are moved as far into goals as possible, such that we have formulas of the following kind only:

`succeeds` *atom*
`fails` *atom*
`terminates (`*goal* `&` ... `&` *goal*`)`

Here is the definition of the functions $S$ and $F$:

$S($`true`$) := $`tt`,
$S($`fail`$) := $`ff`,
$S(s$ `=` $t) := s$ `=` $t$,
$S(A) :=$ `succeeds` $A$,
$S($`~`$G) := F(G)$,
$S(G_1$ `&` ... `&` $G_n) := S(G_1)$ `&` ... `&` $S(G_n)$,
$S(G_1$ `\/` ... `\/` $G_n) := S(G_1)$ `\/` ... `\/` $S(G_n)$.

$F($`true`$) := $`ff`,
$F($`fail`$) := $`tt`,
$F(s$ `=` $t) := s$ `<>` $t$,
$F(A) :=$ `fails` $A$,
$F($`~`$G) := S(G)$,
$F(G_1$ `&` ... `&` $G_n) := F(G_1)$ `\/` ... `\/` $F(G_n)$,
$F(G_1$ `\/` ... `\/` $G_n) := F(G_1)$ `&` ... `&` $F(G_n)$.

Note that $S$ and $F$ are mutually recursive. Here is the definition of $T$:

$T(\texttt{true}) := \texttt{tt}$,
$T(\texttt{fail}) := \texttt{tt}$,
$T(s \texttt{ = } t) := \texttt{tt}$,
$T(A) := \texttt{terminates } A$,
$T(\texttt{~}G) := T(G) \texttt{ \& gr(}x_1\texttt{) \& ... \& gr(}x_n\texttt{)}$, if $\mathrm{FV}(G) = \{x_1, \ldots, x_n\}$,
$T(G_1 \texttt{ \& ... \& } G_n) := \texttt{terminates (}G_1 \texttt{ \& ... \& } G_n\texttt{)}$,
$T(G_1 \texttt{ \textbackslash/ ... \textbackslash/ } G_n) := T(G_1) \texttt{ \& ... \& } T(G_n)$.

Some remarks to the definition of $T$: (i) $T(G)$ expresses universal termination of the goal $G$. This can be seen in the case of disjunctions. A disjunction terminates, if all disjuncts terminate. (ii) The function $T$ does not go into conjunctions. The reason is that we do not want to expand

$$T(G \texttt{ \& } H) := T(G) \texttt{ \& (}F(G) \texttt{ \textbackslash/ } T(H)\texttt{)}.$$

It would lead to an explosion of the size of the formulas. Instead, this equality is built into the inference rules. (iii) In the case of negated goals, the function $T$ expresses that they have to be ground if they are selected. (iv) More motivation for the definition of these functions can be found in [12].

## Definition forms for predicates

Based on the internal database of clauses the LPTP system is able to compute formulas that express success failure and termination of predicates. The functions $S(G)$, $F(G)$ and $T(G)$ are used to compute the definition forms of atoms. In the example of the `member` program (see p. 8) we have:

```
?- needs_gr($(lib)/list/list).

?- def succeeds member(?x,?l1).

(ex l: ?l1 = [?x|?l]) \/
(ex [y,l]: ?l1 = [?y|?l] & succeeds member(?x,?l))

?- def fails member(?x,?l1).

(all l: ~ ?l1 = [?x|?l]) &
(all [y,l]: ?l1 = [?y|?l] => fails member(?x,?l))

?- def terminates member(?x,?l1).

all [y,l]: ?l1 = [?y|?l] => terminates member(?x,?l)
```

Note, that if we instantiate the arguments of the `member/2` predicate with terms, then the defining formulas are automatically simplified:

```
?- def succeeds member(?x,[?y|?l]).
```

```
?x = ?y \/ succeeds member(?x,?l)

?- def fails member(?x,[?y|?l]).

~ ?x = ?y & fails member(?x,?l)

?- def terminates member(?x,[?y|?l]).

terminates member(?x,?l)
```

This automatic simplification is very convenient in practise, since it can reduce, for examle, a disjunction of 10 disjuncts to a simple one line formula in certain cases. In the following we denote by

$D^P$(`succeeds` $A$)
$D^P$(`fails` $A$)
$D^P$(`terminates` $A$)

the results of the commands

```
?- def(succeeds A).
?- def(fails A).
?- def(terminates A).
```

We call these formulas the definition forms for the atom $A$. The exponent $P$ in $D^P$ stands for the logic program under considerations, i.e. the internal database of clauses of LPTP.

## Compiling Prolog programs into the ground representation

As we have seen, in LPTP variables are represented as ground Prolog terms, for example `?x`, `?y`, `?z`. Therefore, before Prolog programs can be verified in LPTP, they have to be translated into a ground representation similar to the one used in the programming language Gödel [5]. The command for compiling a Prolog source file into the ground representation is the following:

```
?- compile_gr(path).
```

This command takes the file *path*`.pl` and compiles it into the file *path*`.gr` which can later be used in the `needs_gr(`*path*`)` command. Note, that the system appends the extensions '`.pl`' and '`.gr`' to the path name automatically.

Prolog variables are translated to LPTP variables in a simple way: the Prolog variable `X1` is translated into `?x1`, the variable `Head` is translated into `?head`, etc. So the system just converts the names to lower case. Anonymous variables (underscores) are translated into `?0`, `?1`, ... and so on. The bodies of clauses are translated into goals in the following way (we write $\bar{G}$ for the translation of $G$):

| | | |
|---|---|---|
| `true` | $\leadsto$ | `true` |
| `fail` | $\leadsto$ | `fail` |
| $s$ `=` $t$ | $\leadsto$ | $\bar{s}$ `=` $\bar{t}$ |
| $r$`(`$t_1$`,...,`$t_n$`)` | $\leadsto$ | $r$`(`$\bar{t_1}$`,...,`$\bar{t_n}$`)` |
| $G_1$ `,` $G_2$ | $\leadsto$ | $\bar{G_1}$ `&` $\bar{G_2}$ |
| $G_1$ `;` $G_2$ | $\leadsto$ | $\bar{G_1}$ `\/` $\bar{G_2}$ |
| `\+` $G$ | $\leadsto$ | `~`$\bar{G}$ |
| `not` $G$ | $\leadsto$ | `~`$\bar{G}$ |
| $G_1$ `->` $G_2$ `;` $G_3$ | $\leadsto$ | `(`$\bar{G_1}$ `&` $\bar{G_2}$`) \/ (~`$\bar{G_1}$ `&` $\bar{G_3}$`)` |

The empty body is translated into the goal `true`.

## Derivations

A derivation is a sequence of derivation steps. Its purpose is to show that a formula necessarily follows from a list of initial assumptions. Here is the grammar of derivations:

*derivation* → *derivation_step*
| `[`*derivation_step*`,...,`*derivation_step*`]`

It is possible to omit the outermost brackets if the derivation constists of a single derivation step only. Derivations are usualy written in vertical mode like this:

`[`*derivation_step*`,`
*derivation_step*`,`
*derivation_step*`]`

In the simplest case, a derivation step is just a formula. In general, a derivation step is a structured object. Here is the grammar for derivation steps:

*derivation_step* → *formula*
| *formula* `by` *tag*
| `assume(`*formula*`,`*derivation*`,`*formula*`)`
| `cases(`*formula*`,`*derivation*`,`*formula*`,`*derivation*`,`*formula*`)`
| `cases([case(`*formula*`,`*derivation*`),...],`*formula*`)`
| `exist(`*name*`,`*formula*`,`*derivation*`,`*formula*`)`
| `exist([`*name*`,...],`*formula*`,`*derivation*`,`*formula*`)`
| `induction([`*formula*`,...],`
`[step([`*name*`,...],[`*formula*`,...],`*derivation*`,`*formula*`),...])`
| `contra(`*formula*`,`*derivation*`)`
| `indirect(~`*formula*`,`*derivation*`)`

This grammar, however, is only an approximation. It does not specify what a correct derivation and what a correct derivation step is. For specifying correct derivations we need more. At every point in a derivation there is a hidden sequence of assumptions $\Gamma$ and a hidden sequence of already derived formulas $\Delta$.

What the proof-checker does is the following: given the sequence of assumptions $\Gamma$ it computes the sequence $\Delta$ and checks at the end whether the formula that the derivation is supposed to prove follows from $\Delta$.

A derivation $d$ is a finite list of derivations steps $[s_1, \ldots, s_n]$. Each derivation step $s_i$ adds exactly one formula $\varphi_i$ to the sequence of allready derived formulas such that the derivation $d$ finally produces the formulas $\varphi_1, \ldots, \varphi_n$. To describe this in a more precise manner we define inductively the following three relations for derivations $d$ and derivation steps $s$.

$\Gamma\{d\}\Delta \rightsquigarrow \Pi$
: Under the assumptions $\Gamma$ the derivation $d$ derives the formulas $\Pi$ from the formulas $\Delta$.

$\Gamma\{s\}\Delta \rightsquigarrow \varphi$
: Under the assumptions $\Gamma$ the derivation step $s$ derives the formula $\varphi$ from the formulas $\Delta$.

$\Gamma; \Delta \triangleright \varphi$
: Under the assumptions $\Gamma$ the formula $\varphi$ is immediately derivable from $\Delta$.

The relations will be correct with respect to the following assertions. We write $\Gamma \models \varphi$ if the formula $\varphi$ is a logical consequence from $\Gamma$ in the sense of classical predicate logic. We write $\Gamma \models \Delta$, if $\Gamma \models \varphi$ for each $\varphi$ in $\Delta$.

1. If $\Gamma\{d\}\Delta \rightsquigarrow \Pi$ and $\Gamma \models \Delta$, then $\Gamma \models \Pi$.
2. If $\Gamma\{s\}\Delta \rightsquigarrow \varphi$ and $\Gamma \models \Delta$, then $\Gamma \models \varphi$.
3. If $\Gamma; \Delta \triangleright \varphi$ and $\Gamma \models \Delta$, then $\Gamma \models \varphi$.

The relation $\Gamma\{d\}\Delta \rightsquigarrow \Pi$ for derivations is defined by the following rule:

$$\frac{\begin{array}{l}\Gamma\{s_1\}\Delta \rightsquigarrow \varphi_1,\\ \Gamma\{s_2\}\Delta, \varphi_1 \rightsquigarrow \varphi_2,\\ \Gamma\{s_3\}\Delta, \varphi_1, \varphi_2 \rightsquigarrow \varphi_3,\\ \quad\vdots\\ \Gamma\{s_n\}\Delta, \varphi_1, \ldots, \varphi_{n-1} \rightsquigarrow \varphi_n,\end{array}}{\Gamma\{[s_1, \ldots, s_n]\}\Delta \rightsquigarrow \Delta, \varphi_1, \ldots, \varphi_n}$$

The derivation step $s_1$ adds the formula $\varphi_1$ to $\Delta$; the derivation step $s_2$ adds the formula $\varphi_2$ to $\Delta, \varphi_1$; and so on. Finally the new formulas that are derived by the derivation $[s_1, \ldots, s_n]$ are $\varphi_1, \ldots, \varphi_n$.

We say that a derivation $d$ is a correct derivation for the formula $\varphi$, if $d$ derives a list of formulas $\Pi$ such that $\varphi$ is immediately derivable from $\Pi$. Formally:

$$\frac{\langle\rangle\{d\}\langle\rangle \rightsquigarrow \Pi \qquad \langle\rangle; \Pi \rightsquigarrow \varphi}{d \text{ is correct for } \varphi}$$

The commands `lemma(`$r$`,`$\varphi$`,`$d$`)`, `theorem(`$r$`,`$\varphi$`,`$d$`)` and `corollary(`$r$`,`$\varphi$`,`$d$`)` check whether $d$ is a correct derivation for $\varphi$.

## 3.2 Derivation steps

In this section we describe the single derivation steps. For each derivation step we give the syntax, the result and a short description. The result of a derivation step is the formula it introduces. The short description contains also the formal definition of the relation $\Gamma\{s\}\Delta \rightsquigarrow \varphi$, where the formula $\varphi$ is the result of the derivation step $s$. We have ordered the derivation steps according to how often they are used. The percent numbers in brackets are based on 46830 lines of formal proofs, or 1.9 MB. The brackets at the top of a description also say whether a derivation step is a pure logical step of classical predicate logic, or whether it is an applied step of the underlying theory of LPTP.

| **Derivation step:** by | 38.4% |
|---|---|

**Syntax:**

*formula* `by` *tag*

**Result:** The step $\varphi$ `by` *tag* introduces the formula $\varphi$.

**Description:** The *tag* gives a hint how the formula $\varphi$ can be derived. The *tag* has to be one of the following:

`theorem(`*reference*`)`
`lemma(`*reference*`)`
`corollary(`*reference*`)`
`axiom(`*reference*`)`
`completion`
`sld`
`introduction(`*name*`,`*arity*`)`
`elimination(`*name*`,`*arity*`)`
`existence(`*name*`,`*arity*`)`
`uniqueness(`*name*`,`*arity*`)`
`builtin`
`concatenation`
`addition`
`gap`
`because`
`[`*tactic*`,...]`

**See also:** Section 3.4, Section 4.1, Section 4.2, Section 4.3.

**Derivation step:** formula 34.1%

**Syntax:**

*formula*

**Result:** The trivial step $\varphi$ introduces the formula $\varphi$.

**Description:** In the simplest case a derivation step is just a formula $\varphi$ that is immediately derivable from the already derived formulas $\Delta$. The formula $\varphi$ is then added to $\Delta$.

**Formal rule:**

$$\frac{\Gamma; \Delta \triangleright \varphi}{\Gamma\{\varphi\}\Delta \rightsquigarrow \varphi}$$

**Example:** In the simplest case a derivation is just a sequence of formulas. Assume that the following formula belongs to $\Delta$:

```
all [x,l]:terminates r(?x,?l)
```

Then the following is a correct derivation:

```
[terminates r(?x,?l),
 succeeds r(?x,?l) \/ fails r(?x,?l),
 ...]
```

We have used the following inference rules for immediate consequence:

```
all [x,l]:terminates r(?x,?l) ▷ terminates r(?x,?l)
terminates r(?x,?l) ▷ succeeds r(?x,?l) \/ fails r(?x,?l)
```

**See also:** Section 3.3.

**Derivation step:** assume (implication introduction) 14.3%

**Syntax:**

`assume(`*formula*`,`
*derivation*`,`
*formula*`)`

**Result:** The step `assume(`$\varphi$`,`$d$`,`$\psi$`)` yields the formula $\varphi$ `=>` $\psi$.

**Description:** In order to prove $\varphi$ `=>` $\psi$ one assumes that $\varphi$ is true and derives $\psi$ from $\varphi$. This means that the formula $\varphi$ is added to the assumptions $\Gamma$ and to the already derived formulas $\Delta$. The derivation $d$ then must derive a sequence of formulas $\Pi$ from $\Gamma$ and $\Delta$ such that $\psi$ is an immediate consequence of $\Pi$.

**Formal rule:**

$$\frac{\Gamma, \varphi\{d\}\Delta, \varphi \rightsquigarrow \Pi \qquad \Gamma; \Pi \triangleright \psi}{\Gamma\{\texttt{assume(}\varphi\texttt{,}d\texttt{,}\psi\texttt{)}\}\Delta \rightsquigarrow \varphi \texttt{ => } \psi}$$

**Example:** In the following lemma we assume that `p & q` is true. Then we infer as immediate consequences the formulas `p`, `q` and `q & p`.

```
assume(p & q,
 [p, q, q & p],
 q & p)
```

The formula which is introduced is `p & q => q & p`.

**Derivation step:** cases (disjunction elimination) 5.3%

**Syntax:**

```
cases(formula,
 derivation,
 formula,
 derivation
 formula
)

cases([
 case(formula,
  derivation),
 ...],
 formula)
```

**Result:** The step `cases(`$\varphi_1$`,`$d_1$`,`$\varphi_2$`,`$d_2$`,`$\psi$`)` introduces the formula $\psi$. The step `cases([case(`$\varphi$`,`$d$`),...],`$\psi$`)` introduces the formula $\psi$.

**Description:** Assume that we have a disjunction $\varphi_1$ `\/ ... \/` $\varphi_n$ that follows immediately from $\Delta$ and that we want to prove the formula $\psi$. We can prove the formula $\psi$ by a case splitting. In each case we assume that the formula $\varphi_i$ is true, i.e. we add $\varphi_i$ temporarily to $\Gamma$ and $\Delta$, and derive with the derivation $d_i$ formulas $\Pi_i$ such that $\psi$ follows immediately from $\Pi_i$. Then we can infer $\psi$.

**Formal rule:**

$$\frac{\Gamma;\Delta \triangleright \varphi_1 \ \texttt{\textbackslash/}\ \ldots\ \texttt{\textbackslash/}\ \varphi_n \quad \Gamma,\varphi_i\{d_i\}\Delta,\varphi_i \rightsquigarrow \Pi_i \quad \Gamma,\varphi_i;\Pi_i \triangleright \psi \quad \text{for } i=1,\ldots,n}{\Gamma\{\texttt{cases([case(}\varphi_1\texttt{,}d_1\texttt{),...,case(}\varphi_n\texttt{,}d_n\texttt{)],}\psi\texttt{)}\}\Delta \rightsquigarrow \psi}$$

In the special case of $n = 2$ we have:

$$\frac{\Gamma;\Delta \triangleright \varphi_1 \ \texttt{\textbackslash/}\ \varphi_2 \quad \Gamma,\varphi_i\{d_i\}\Delta,\varphi_i \rightsquigarrow \Pi_i \quad \Gamma,\varphi_i;\Pi_i \triangleright \psi \quad \text{for } i=1,2}{\Gamma\{\texttt{cases(}\varphi_1\texttt{,}d_1\texttt{,}\varphi_2\texttt{,}d_2\texttt{,}\psi\texttt{)}\}\Delta \rightsquigarrow \psi}$$

**Example:** In the following example we make a case splitting for `p \/ q`.

```
assume((p \/ q) & r,
 cases(p, p & r,
  q, q & r,
  p & r \/ q & r),
 p & r \/ q & r)
```

The case step introduces `(p \/ q) & r => p & r \/ q & r`.

**See also:** Tactic `case`, p. 109.

**Derivation step:** exist (exist elimination) 4.8%

**Syntax:**

```
exist(name,formula,
 derivation,
 formula
)

exist([name,...],formula,
 derivation,
 formula
)
```

**Result:** Both, the step `exist(`$x$`,`$\varphi$`,`$d$`,`$\psi$`)` and `exist([`$x$`,...],`$\varphi$`,`$d$`,`$\psi$`)` introduce the formula $\psi$.

**Description:** Assume that we have an existentially quantified formula `ex` $x$`:`$\varphi$ which is immediately derivable from $\Delta$. Suppose we want to prove the formula $\psi$. We say, let $x$ be such that $\varphi$ is true, i.e. we add $\varphi$ to $\Gamma$ and $\Delta$, and derive then a sequence of formulas $\Pi$ such that $\psi$ is derivable from $\Pi$ immediately. If the variable $x$ does not occur free in $\psi$, i.e. if $\psi$ is independent of the choice of $x$, we can say that it follows from `ex` $x$`:`$\varphi$.

**Formal rule:**

$$\frac{\Gamma;\Delta \triangleright \texttt{ex}\ x\texttt{:}\varphi \qquad \Gamma,\varphi\{d\}\Delta,\varphi \rightsquigarrow \Pi \qquad \Gamma;\Pi \triangleright \psi \quad x \notin \mathrm{FV}(\psi)}{\Gamma\{\texttt{exist(}x\texttt{,}\varphi\texttt{,}d\texttt{,}\psi\texttt{)}\}\Delta \rightsquigarrow \psi}$$

In the general case with more than one quantified variable the rule is:

$$\frac{\Gamma;\Delta \triangleright \texttt{ex}\ \texttt{[}x\texttt{,...]:}\varphi \qquad \Gamma,\varphi\{d\}\Delta,\varphi \rightsquigarrow \Pi \qquad \Gamma;\Pi \triangleright \psi \quad [x,\ldots] \cap \mathrm{FV}(\psi) = \emptyset}{\Gamma\{\texttt{exist([}x\texttt{,...],}\varphi\texttt{,}d\texttt{,}\psi\texttt{)}\}\Delta \rightsquigarrow \psi}$$

**Example:** Note, that LPTP considers formulas equivalent modulo renaming of bound variables.

```
assume(ex x:r(?x) & p,
 exist(y,r(?y),
  ex x:r(?x),
  (ex x:r(?x)) & p),
 (ex x:r(?x)) & p)
```

The derivations step introduces `(ex x:r(?x) & p) => (ex x:r(?x)) & p`.

**See also:** Tactic `ex`, p. 111.

**Derivation step:** induction 2.4%

**Syntax:**

```
induction([formula,...],
 [step([name,...],
   [formula,...],
   derivation,
   formula),
  ...])
```

**Result:** The derivation step `induction([`$\varphi_1$`,...,`$\varphi_n$`],`*induction_steps*`)` introduces the formula $\varphi_1$ `&` ... `&` $\varphi_n$.

**Description:** Induction means simultaneous induction along the definitions of predicates in a logic program. Induction has the form:

`induction([`$\varphi_1$`,...,`$\varphi_n$`],`*induction_steps*`)`

The formulas $\varphi_1, \ldots, \varphi_n$ are the formulas that have to be derived using induction. Each formula $\varphi_i$ must be of the following form:

`all [`$x_1$`,...,`$x_n$`]: succeeds` $R$`(`$x_1$`,...,`$x_n$`) =>` $\psi$

The precicate symbol $R$ must have clauses that define it in the current database of clauses. An example, where $n = 1$, is the following:

```
induction(
 [all l: succeeds list(?l) => terminates member(?x,?l)],
 induction_steps)
```

In this example we want to proof by induction on `list(?l)` that `member(?x,?l)` terminates. Since the predicate `list` has two clauses (see p. 8) there are two induction steps. In general, an induction step has the following form:

`step([`$x_1$`,...,`$x_m$`],`
`[`$\psi_1$`,...,`$\psi_k$`],`
$d$`,`
$\chi$`)`

The formula $\psi_1, \ldots, \psi_k$ are the induction hypotheses and the formula $\chi$ is the conclusion that has to be derived by $d$ from the hypotheses. The number and the structure of the formulas $\psi_j$ depend on the structure of the clauses in the program. The induction step derives the following formula:

`all [`$x_1$`,...,`$x_m$`]:` $\psi_1$ `&` ... `&` $\psi_k$ `=>` $\chi$

**Formal rule:**

$$\frac{\Gamma, \vec{\psi}\{d\}\Delta, \vec{\psi} \rightsquigarrow \Pi \qquad \Gamma; \Pi \triangleright \chi \qquad \{\vec{x}\} \cap \mathrm{FV}(\Gamma) = \emptyset}{\Gamma\{\texttt{step([}\vec{x}\texttt{],[}\vec{\psi}\texttt{],}d\texttt{,}\chi\texttt{)}\}\Delta \rightsquigarrow \texttt{all [}\vec{x}\texttt{]:}\vec{\psi}\ \texttt{=>}\ \chi}$$

The bound variables $\vec{x}$ can be changed by the user. So one could as well use the induction step `step([`$\vec{y}$`],[`$\vec{\psi'}$`],`$d'$`,`$\chi'$`)` where $\psi'$, $d'$, $\chi'$ are obtained from $\psi$, $d$, $\chi$ by renaming $\vec{x}$ to $\vec{y}$. In the example of `list` there are two induction steps. The first step is:

```
step([],
 [],
 d,
 terminates member(?x,[]))
```

There are no induction hypotheses in this case and the conclusion that has to be derived by $d$ is the formula `terminates member(?x,[])`. A better name for this kind of step would be *base case*. The second induction step is:

```
step([y,l],
 [terminates member(?x,?l), succeeds list(?l)],
 d,
 terminates member(?x,[?y|?l]))
```

There are two induction hypothesis here:

```
terminates member(?x,?l) & succeeds list(?l)
```

The conclusion that has to be derived by $d$ is the formula

```
terminates member(?x,[?y|?l]).
```

Here is the full induction proof of the example:

```
induction(
 [all l: succeeds list(?l) => terminates member(?x,?l)],
  [step([],
    [],
    terminates member(?x,[]) by completion,
    terminates member(?x,[])),
   step([y,l],
    [terminates member(?x,?l),
     succeeds list(?l)],
    terminates member(?x,[?y|?l]) by completion,
    terminates member(?x,[?y|?l]))])]
)
```

In the simple case of induction on `list` the induction scheme may look a little bit clumsy. However, it has been designed for mutual recursive predicates and in practical application it soon happens that an induction has 44 cases (see Chapter 7). The LPTP system knows how to create the induction scheme and so the user does not have to worry about it. In our example, the induction scheme can be created by the following tactic:

```
?- needs_gr($(lib)/list/list).
?- all l: succeeds list(?l) => terminates member(?x,?l)
    by [ind].
```

**See also:** Tactic `ind` on p. 113 and tactic `indqf` on p. 114. For the theoretical foundations of the induction scheme, see [13].

**Derivation step:** contra (proof by contradiction) 0.5%

**Syntax:**

`contra(`*formula*`,`
*derivation*`)`

**Result:** The step `contra(`$\varphi$`,`$d$`)` introduces the formula `~`$\varphi$.

**Description:** This derivation step corresponds to a proof by contradiction. Assume that we want to prove the formula `~`$\varphi$. We assume $\varphi$, i.e. we add $\varphi$ to $\Gamma$ and $\Delta$ and try to derive a contradiction. The derivation $d$ must derive a sequence of formulas $\Pi$ which is inconsistent. This means that the constant `ff` (falsum) is immediately derivable from $\Pi$.

**Formal rule:**

$$\frac{\Gamma, \varphi\{d\} \rightsquigarrow \Pi \qquad \Gamma; \Pi \triangleright \texttt{ff}}{\Gamma\{\texttt{contra(}\varphi\texttt{,}d\texttt{)}\}\Delta \rightsquigarrow \texttt{~}\varphi}$$

**Example:** The following example derives `~p & ~q => ~(p \/ q)`.

```
assume(~p & ~q,
 contra(p \/ q,
  cases(p,ff,q,ff,ff),
  ff),
 ~(p \/ q))
```

**Derivation step:** indirect (indirect proof) 0.1%

**Syntax:**

`indirect(~`*formula*`,`
*derivation*`)`

**Result:** The step `indirect(~`$\varphi$`,`$d$`)` introduces the formula $\varphi$.

**Description:** This derivation step corresponds to the principle of indirect proofs. This principle is only valid in classical logic and not in intuitionistic logic. Since LPTP is classical system, the principle is available in LPTP. Suppose that we want to prove the formula $\varphi$. We assume the negation of $\varphi$ and add it to $\Gamma$ and $\Delta$ and try to derive a contradiction. The derivation $d$ must derive an inconsistent sequence of formulas $\Pi$ from which the constant `ff` (falsum) is immediately derivable.

**Formal rule:**

$$\frac{\Gamma, \tilde{}\varphi\{d\}\Delta, \tilde{}\varphi \rightsquigarrow \Pi \qquad \Gamma; \Pi \rightsquigarrow \texttt{ff}}{\Gamma\{\texttt{indirect(}\tilde{}\varphi\texttt{,}d\texttt{)}\}\Delta \rightsquigarrow \varphi}$$

**Example:** The following derivation derives `~(p & q) => ~p \/ ~q`.

```
assume(~(p & q),
 indirect(~(~p \/ ~q),
  [indirect(~p,[~p \/ ~q, ff]),
   indirect(~q,[~p \/ ~q, ff]),
   p & q, ff]),
 ~p \/ ~q)
```

Note, that the formula `~(p & q) => ~p \/ ~q` is intuitionistically not valid. The derivation step *indirect* is not the only reason that LPTP is classical system. For example, the formula $\varphi$ `\/ ~`$\varphi$ (tertium non datur) is built into LPTP as an axiom.

## 3.3 Inference rules

In this section we define the relation $\Gamma; \Delta \triangleright \varphi$ which expresses that the formula $\varphi$ is immediately derivable from the sequence of formulas $\Delta$ under the assumptions of $\Gamma$. The purpose of $\Gamma$ is to ensure that its free variables are not bound in an immediate derivation. Remember that the soundness assertion for $\Gamma; \Delta \triangleright \varphi$ is the following: if $\Gamma \models \Delta$ then $\Delta \models \varphi$.

The relation $\Gamma; \Delta \triangleright \varphi$ for immediate derivation is used in its pure form in the simple derivation steps that consists of single formulas only. Remember that this derivation step is used in 34% of all derivation steps. The numbers in this section now say how often each single case of an immediate derivation is used. We want to point out the 0.0% does not necessarily mean that the rules has been never used. However, the alpha-conversion rule below is really never used. Perhaps this is because alpha-conversion (renaming of bound variables) is built into the other rules. Every rule that uses matching or equality does automatically include alpha-conversion.

Immediate derivations are not that simple as the name might suggest. The relation $\Gamma; \Delta \triangleright \varphi$ is defined in terms of a more refined relation $\Gamma; \Delta \triangleright^n \varphi$ that includes a depth $n$. In fact, at the moment LPTP defines the relation $\Gamma; \Delta \triangleright \varphi$ as follows:

$$\frac{\Gamma; \Delta \triangleright^6 \varphi}{\Gamma; \Delta \triangleright \varphi.}$$

The number 6 is called the *depth of thinking* of LPTP and this number could be changed in the future.

Remember that a capital Greek letter like $\Delta$ denotes finite list of formulas. $\Delta$ has to be understood as the conjunction of its elements. If one of the elements of $\Delta$ is itself a conjunction then we consider the conjuncts also as elements of $\Delta$. In other words, if we write $\varphi$ *in* $\Delta$ below, then we mean that

1. $\varphi$ is an element of $\Delta$, or
2. there exists an element $\psi_1$ `&` ... `&` $\psi_n$ of $\Delta$ and an $i$, $1 \leq i \leq n$, such that $\varphi$ is $\psi_i$.

For example, we have:

```
r(?x) in [?x = ?y, succeeds list(?x) & r(?x)]
```

In this way we avoid an explicit conjunction elimination rule or, more precisely, we include it into the identity rule. Note also, that we may assume that the $\psi_i$'s are not conjunctions, since otherwise they would have been flattened automatically. Therefore, we do not have to iterate the process in the definition of *in*.

### Substitutions

A substitution $\sigma$ is considered as a finite set ob bindings:

$$\sigma = \{t_1/x_1, \ldots, t_n/x_n\}.$$

It is not required that the term $t_i$ are different from the variable $x_i$. However, the variables $x_1, \ldots, x_n$ have to be pairwise different. We define

$$spt(\sigma) := \{x_i \mid 1 \leq i \leq n,\ x_i \not\equiv t_i\}.$$

We write $\varphi\sigma$ for the result of applying the substitution $\sigma$ to the formula $\varphi$. This means that the variables $x_i$ are simultaneously replaced by the terms $t_i$ and the bound variables are renamed when necessary.

| **Inference rule:** identity | 44.8% |
|---|---|

**Description:** This is the most elementary and also by far most used inference rule. It says that we can infer a formula $\varphi$ if it belongs already to $\Delta$.

**Formal rule:**

$$\frac{\varphi \ in \ \Delta}{\Gamma; \Delta \rhd^n \varphi}$$

| **Inference rule:** modus ponens (matching) | 16.1% |
|---|---|

**Description:** This rule is a combination of instantiation of universal quantifiers, modus ponens and elimination of conjunctions. It includes the following rules:

- From `(all [`$\vec{x}$`]:` $\varphi(\vec{x})$`)` we can infer $\varphi(\vec{t})$.
- From `(all [`$\vec{x}$`]:` $\varphi_1(\vec{x})$ `&` ... `&` $\varphi_n(\vec{x})$`)` we can infer $\varphi_i(\vec{t})$.
- From `(all [`$\vec{x}$`]:` $\varphi(\vec{x})$ `=>` $\psi(\vec{x})$`)` and $\varphi(\vec{t})$ we can infer $\psi(\vec{t})$.

It includes also more complicated rules. For example, from

`(all [`$\vec{x}$`]:` $\varphi(\vec{x})$ `=> (all [`$\vec{y}$`]:` $\psi(\vec{x},\vec{y})$ `=>` $\chi(\vec{x},\vec{y})$`))`

and $\varphi(\vec{s})$ and $\psi(\vec{s},\vec{t})$ we can infer $\chi(\vec{s},\vec{t})$.

**Formal rules:**

$$\frac{mp(\Delta,\varphi)}{\Gamma;\Delta \rhd^n \varphi} \qquad \frac{\varphi \ in \ \Delta}{mp(\Delta,\varphi)} \qquad \frac{mp(\Delta, \varphi_1 \ \texttt{\&} \ \ldots \ \texttt{\&} \ \varphi_n)}{mp(\Delta,\varphi_i)}$$

$$\frac{mp(\Delta, \varphi_1 \ \texttt{\&} \ \ldots \ \texttt{\&} \ \varphi_n \ \texttt{=>} \ \psi) \qquad \varphi_1 \ in \ \Delta \quad \ldots \quad \varphi_n \ in \ \Delta}{mp(\Delta,\psi)}$$

$$\frac{mp(\Delta, \texttt{(all [}\vec{x}\texttt{]:} \ \varphi\texttt{)}) \qquad spt(\sigma) \subseteq \{\vec{x}\}}{mp(\Delta,\varphi\sigma)}$$

**Example:** From the list of formulas

```
all [x,y,z]: r(?x,?y) & r(?y,?z) => r(?x,?z)
r(1,2), r(2,3)
```

we can infer r(1,3).

**Example:** From `(all [x,y]: r(?x,?y))` we can infer `r(1,2)`.

**Inference rule:** conjunction introduction 7.4%

**Description:** If we can derive $\varphi_i$ from $\Delta$ with length $n$ for each $i = 1, \ldots, k$, then we can derive the conjunction $\varphi_1$ `&` ... `&` $\varphi_k$ with length $n + 1$.

**Formal rule:**

$$\frac{\Gamma; \Delta \triangleright^n \varphi_i \quad \text{for } i = 1, \ldots, k}{\Gamma; \Delta \triangleright^{n+1} \varphi_1 \texttt{ \& ... \& } \varphi_k}$$

**Example:** From

```
p, p => q
```

we can infer

```
p & q.
```

**Inference rule:** forall introduction 7.1%

**Description:** If $\varphi$ is derivable from $\Delta$ with length $n$ and the variables $\vec{x}$ do not appear free in any formula of $\Gamma$, then we can infer `(all [`$\vec{x}$`]:` $\varphi$`)` with length $n + 1$.

**Formal rule:**

$$\frac{\Gamma; \Delta \triangleright^n \psi \qquad \mathrm{FV}(\Gamma) \cap \{\vec{x}\} = \emptyset}{\Gamma; \Delta \triangleright^{n+1} \texttt{(all [}\vec{x}\texttt{]: } \varphi\texttt{)}}$$

**Example:** If the variables `?x` and `?l` do not appear free in any assumption, we can infer from

```
list(?l) => terminates member(?x,?l)
```

the formula

```
all [x,l]: list(?l) => terminates member(?x,?l).
```

| **Inference rule:** unification (CET) | 5.5% |
|---|---|

**Description:** Suppose that we want to derive an equation $s = t$. First, we collect all the equations of $\Delta$ into a set $E$. If $E$ is not unifiable, then we already have a contradiction. Otherwise, let $\sigma$ be the most general unifier of $E$. If $s\sigma \equiv t\sigma$, then we can infer $s = t$.

To infer a formula $\varphi$, we can proceed in the same way. If $\sigma$ is a most general unifier of $E$ and $\psi$ is in $\Delta$ and $\varphi\sigma \equiv \psi\sigma$, then we can infer $\varphi$.

**Formal rules:**

$$\frac{\mathit{pure\_equations}(\Delta) = E \qquad E \quad \text{not unifiable}}{\Gamma; \Delta \rhd^n \varphi}$$

$$\frac{\mathit{pure\_equations}(\Delta) = E \qquad \mathit{mgu}(E) = \sigma \qquad s\sigma \equiv t\sigma}{\Gamma; \Delta \rhd^n s = t}$$

$$\frac{\mathit{pure\_equations}(\Delta) = E \qquad \mathit{mgu}(E) = \sigma \qquad \psi \ \mathit{in}\ \Delta \qquad \varphi\sigma \equiv \psi\sigma}{\Gamma; \Delta \rhd^n \varphi}$$

By $\mathit{pure\_equations}(\Delta)$ we denote the set of all equations $s = t$ which are in $\Delta$ such that $s$ and $t$ are pure constructor terms which do not contain defined function symbols (see Section 4.2).

**Remark:** This simple and efficient treatment of equality is based on the so-called *Clark equality theory CET* (see [3]). This equality theory axiomatizes unification with following formulas:

1. $x$ `=` $y$
2. $x$ `=` $y$ `&` $y$ `=` $z$ `=>` $x$ `=` $z$
3. $x$ `=` $y$ `=>` $y$ `=` $x$
4. $x_1$ `=` $y_1$ `&` `...` `&` $x_n$ `=` $y_n$ `=>` $f$`(`$x_1$`,...,`$x_n$`)` `=` $f$`(`$y_1$`,...,`$y_n$`)`
5. $f$`(`$x_1$`,...,`$x_n$`)` `=` $f$`(`$y_1$`,...,`$y_n$`)` `=>` $x_i$ `=` $y_i$
6. $f$`(`$x_1$`,...,`$x_n$`)` `<>` $g$`(`$y_1$`,...,`$y_m$`)`, if $f$ is different from $g$,
7. $t$ `<>` $x$, if the variable $x$ occurs in $t$ and $t$ is different from $x$
8. $x_1$ `=` $y_1$ `&` `...` `&` $x_n$ `=` $y_n$ `&` $R$`(`$x_1$`,...,`$x_n$`)` `=>` $R$`(`$y_1$`,...,`$y_n$`)`

**Example:** From the list of equations

```
[?x1|?l1] = [?x2|?l2], s(?y) = s(?x2)
```

we can infer

```
?x1 = ?y.
```

**Inference rule:** sld step 2.5%

**Description:** This rule is the same as the following derivation step.

```
succeeds R(t1,...,tn) by sld
```

See p. 82 for more details.

**Formal rule:**

$$\frac{(A \texttt{ :- } G) \in \mathit{Clauses} \qquad \mathit{body}(\Delta, S(G\sigma))}{\Gamma; \Delta \rhd^n \texttt{succeeds } A\sigma}$$

**Example:** From the formula

```
succeeds member(?x,[?v|?l])
```

we can infer

```
succeeds member(?x,[?u,?v|?l]).
```

**See also:** See p. 46 for the definition of the formula $S(G)$ and p. 82 for the definition of $\mathit{body}(\Delta, \varphi)$.

**Inference rule:** disjunction introduction 2.4%

**Description:** A disjunction $\varphi_1$ `\/` ... `\/` $\varphi_m$ can be inferred with length $n+1$ if there exists an $i$, $1 \leq i \leq m$, such that $\varphi_i$ can be inferred with length $n$.

**Formal rule:**

$$\frac{\Gamma; \Delta \rhd^n \varphi_i}{\Gamma; \Delta \rhd^{n+1} \varphi_1 \texttt{ \textbackslash/ } \ldots \texttt{ \textbackslash/ } \varphi_m}$$

**Example:** From `p` we can infer `p \/ q`.

**See also:** The generalized disjunction introduction on p. 77.

| **Inference rule:** inconsistency | 2.2% |
|---|---|

**Description:** If $\Delta$ is inconsistent, then we can infer $\varphi$. A sequence $\Delta$ is inconsistent, if one of the following holds:

1. `ff` is in $\Delta$ [43.0%]
2. `succeeds` $G$ and `fails` $G$ are in $\Delta$ [32.9%]
3. $\psi$ and `~`$\psi$ are in $\Delta$ [14.4%]
4. $s$ `=` $t$ and $s$ `<>` $t$ are in $\Delta$ [8.3%]
5. $t$ `<>` $t$ is in $\Delta$ [1.3%]

**Formal rules:**

$$\frac{\texttt{ff}\ \mathit{in}\ \Delta}{\Gamma; \Delta \rhd^n \varphi} \qquad \frac{\psi\ \mathit{in}\ \Delta \qquad \texttt{\~{}}\psi\ \mathit{in}\ \Delta}{\Gamma; \Delta \rhd^n \varphi} \qquad \frac{s\ \texttt{=}\ t\ \mathit{in}\ \Delta \qquad s\ \texttt{<>}\ t\ \mathit{in}\ \Delta}{\Gamma; \Delta \rhd^n \varphi}$$

$$\frac{t\ \texttt{<>}\ t\ \mathit{in}\ \Delta}{\Gamma; \Delta \rhd^n \varphi} \qquad \frac{\texttt{(succeeds}\ G\texttt{)}\ \mathit{in}\ \Delta \qquad \texttt{(fails}\ G\texttt{)}\ \mathit{in}\ \Delta}{\Gamma; \Delta \rhd^n \varphi}$$

| **Inference rule:** equality | 2.0% |
|---|---|

**Description:** This is the standard equality rule that says from $s$ `=` $t$ and $\varphi(s)$ we can infer $\varphi(t)$. The LPTP rule, however, is a little bit more general. In LPTP an equation is not only an equation between *two* terms but an equation between *several* terms, written as $t_1$ `=` $t_2$ `=` `...` `=` $t_n$. Moreover, symmetry is also included into the equality rule.

Let $E$ be the set of (general) equations of $\Delta$. We say that two terms $s$ and $t$ are syntactically equal modulo $E$, if $t$ can be obtained from $s$ by replacing subterms $r_1, \ldots, r_n$ by terms $r'_1, \ldots, r'_n$ such that for each $i = 1, \ldots, n$ there is an equation in $E$ that states that $r_i$ and $r'_i$ are equal. In the same way we define that a formula $\varphi$ is equal to $\psi$ modulo $E$.

Now, an equation $t_1$ `=` $t_2$ `=` `...` `=` $t_n$ can be inferred, if $t_i$ is equal to $t_{i+1}$ modulo $E$ for $i = 1, \ldots, n-1$. A formula $\varphi$ can be inferred, if there exists a formula $\psi$ in $\Delta$ such that $\varphi$ is equal to $\psi$ modulo $E$.

**Formal Rules:**

$$\frac{\mathit{equations}(\Delta) = E \qquad t_1 \equiv t_{i+1} \pmod{E} \quad \text{for } i = 1, \ldots, n-1}{\Gamma; \Delta \rhd^n t_1 \texttt{ = } \ldots \texttt{ = } t_n}$$

$$\frac{\mathit{equations}(\Delta) = E \qquad \psi \mathit{ in } \Delta \qquad \varphi \equiv \psi \pmod{E}}{\Gamma; \Delta \rhd^n \varphi}$$

**Example:** From the formulas

```
?x = ?y, r(s(?x),?y)
```

we can infer

```
r(s(?y),?x).
```

From the list of equations

```
s(?x) + ?y = s(?x + ?y)
?x + ?y = ?y + ?x
s(?y + ?x) = ?y + s(?x)
```

we can infer

```
s(?x) + ?y = s(?x + ?y) = s(?y + ?x) = ?y + s(?x).
```

| **Inference rule:** atom introduction (user-defined) | 1.5% |
|---|---|

**Description:** This rule is the same as the following derivation steps:

```
succeeds atom by completion
fails atom by completion
terminates atom by completion
```

It says that if $\varphi$ is the defining formula for `succeeds` $A$ and $\varphi$ can be inferred with length $n$ then `succeeds` $A$ can be inferred with length $n+1$. The same rule can be applied with the defining formulas for `fails` $A$ and `terminates` $A$. See p. 83 for more details.

**Formal rules:**

$$\frac{\Gamma;\Delta \rhd^n D^P(\texttt{succeeds}\ A)}{\Gamma;\Delta \rhd^{n+1} \texttt{succeeds}\ A}$$

$$\frac{\Gamma;\Delta \rhd^n D^P(\texttt{fails}\ A)}{\Gamma;\Delta \rhd^{n+1} \texttt{fails}\ A} \qquad \frac{\Gamma;\Delta \rhd^n D^P(\texttt{terminates}\ A)}{\Gamma;\Delta \rhd^{n+1} \texttt{terminates}\ A}$$

**Example:** From

```
?x <> ?y
fails member(?x,?l)
```

we can infer

```
fails member(?x,[?y|?l]).
```

| **Inference rule:** modus ponens (plain) | 1.3% |
|---|---|

**Description:** If $\psi$ `=>` $\varphi$ is in $\Delta$ and $\psi$ is derivable with length $n$, then $\varphi$ is derivable with length $n+1$.

**Formal rule:**

$$\frac{(\psi\ \texttt{=>}\ \varphi)\ in\ \Delta \qquad \Gamma;\Delta \rhd^n \psi}{\Gamma;\Delta \rhd^{n+1} \varphi}$$

**Example:** From

```
p, p => q
```

we can infer `q`.

| **Inference rule:** special axiom | 1.1% |
|---|---|

**Description:** A formula $\varphi$ can be inferred, if it is a special axiom. Special axioms are:

1. $t$ `=` $t$ [49.1%]
2. `gr(`$c$`)`, if $c$ is a constant, [20.2%]
3. $f$`(`$s_1$`,...,`$s_m$`)` `<>` $g$`(`$t_1$`,...,`$t_n$`)`, if $f \neq g$ or $m \neq n$, [11.0%]
4. $\varphi$ `\/` `~`$\varphi$, $s$ `=` $t$ `\/` $s$ `<>` $t$ [9.5%]
5. `~(`$s$ `=` $t$`)`, if $s$ and $t$ are pure and not unifiable, [9.5%]
6. $s$ `<>` $t$, if $s$ and $t$ are pure and not unifiable, [0.6%]
7. $\varphi$ `<=>` $\varphi$ [0.0%]

**Formal rule:**

$$\frac{\varphi \quad \text{a special axiom}}{\Gamma; \Delta \rhd^n \varphi}$$

**Remark:** The special axioms are redundant. They have been added for efficiency reasons and in order to shorten formal proofs.

**Inference rule:** modus ponens (including derivability) 1.0%

**Description:** The generalized modus ponens rule says the following. If the formula

`all [`$\vec{x}$`]:`$\varphi(\vec{x})$ `=>` $\psi(\vec{x})$

is in $\Delta$ and $\varphi(\vec{t})$ is derivable from $\Delta$ with length $n$ then $\psi(\vec{t})$ is derivable with length $n+1$.

**Formal rule:**

$$\frac{\texttt{(all [}\vec{x}\texttt{]:}\varphi \texttt{ => } \psi\texttt{)}\ \mathit{in}\ \Delta \qquad \mathit{spt}(\sigma) \subseteq \{\vec{x}\} \cap \mathrm{FV}(\psi) \qquad \Gamma; \Delta \rhd^n \varphi\sigma}{\Gamma; \Delta \rhd^{n+1} \psi\sigma}$$

**Example:** From

```
all [x,l]: succeeds list(?l) => terminates member(?x,?l)
succeeds list(?l)
```

we can infer

```
terminates member(?x,?l).
```

**Remark:** Note, that the formal rule does not say exactly the same as the informal description. In the informal description one has to require in addition that the variables $\vec{x}$ actually occur in the formula $\chi$.

**See also:** Modus ponens with matching on p. 64.

**Inference rule:** exist introduction 0.9%

**Description:** From $\varphi(\vec{t})$ we can infer the formula `(ex [`$\vec{x}$`]:` $\varphi(\vec{x})$`)`.

**Formal rule:**

$$\frac{\varphi\sigma\ \mathit{in}\ \Delta \qquad \mathit{spt}(\sigma) \subseteq \{\vec{x}\}}{\Gamma; \Delta \rhd^n \texttt{(ex [}\vec{x}\texttt{]:}\ \varphi\texttt{)}}$$

**Example:** From `r(1,2)` we can infer

```
ex [x,y]: r(?x,?y).
```

**Inference rule:** termination introduction (termination) 0.8%

**Description:** If $T(G)$ and $T(H_1$ `&` ... `&` $H_m)$ are derivable with length $n$ then the formula

`terminates (`$G$ `&` $H_1$ `&` ... `&` $H_m$`)`

is derivable with length $n+1$. Here, $T$ is the built-in function that associates to a goal a formula that expresses its termination (see p. 46).

**Formal rule:**

$$\frac{\Gamma;\Delta \rhd^n T(G) \qquad \Gamma;\Delta \rhd^n T(H_1 \texttt{ \& ... \& } H_m)}{\Gamma;\Delta \rhd^{n+1} \texttt{terminates (}G \texttt{ \& } H_1 \texttt{ \& ... \& } H_m\texttt{)}}$$

**Example:** From

```
terminates member(?x,?l1)
gr(?x) & gr(?l2)
terminates member(?x,?l2)
```

we can infer

```
terminates (member(?x,?l1) & ~member(?x,?l2)).
```

**See also:** Tactic `unfold` on p. 115.

**Inference rule:** exist introduction (matching) 0.7%

**Description:** If $\varphi_i(\vec{t})$ is in $\Delta$ for $i = 1, \ldots, m$ then we can infer the formula

`ex [`$\vec{x}$`]:` $\varphi_1(\vec{x})$ `&` ... `&` $\varphi_n(\vec{x})$`.`

**Formal rule:**

$$\frac{\varphi_1\sigma \textit{ in } \Delta \quad \ldots \quad \varphi_n\sigma \textit{ in } \Delta \qquad \textit{spt}(\sigma) \subseteq \{\vec{x}\}}{\Gamma;\Delta \rhd^n \texttt{ex [}\vec{x}\texttt{]: } \varphi_1 \texttt{ \& ... \& } \varphi_n}$$

**Example:** From `p(2)` and `q(2,3)` we can infer

```
ex [x,y]: p(?x) & q(?x,?y).
```

| **Inference rule:** ground introduction (variable) | 0.6% |
|---|---|

**Description:** If there exists a formula `gr(`$f$`(`$t_1$`,...,`$t_m$`))` in $\Delta$ and an index $i$ such that

- $1 \leq i \leq m$,
- $t_i$ is a pure constructor term,
- and the variable $x$ occurs in $t_i$,

then we can infer the formula `gr(`$x$`)`.

**Formal rule:**

$$\frac{\texttt{gr(}f\texttt{(}t_1\texttt{,...,}t_m\texttt{))}\ \mathit{in}\ \Delta \qquad t_i \in \mathit{Pure\_Terms} \qquad x \in \mathrm{FV}(t_i)}{\Gamma; \Delta \rhd^n \texttt{gr(}x\texttt{)}}$$

**Example:** From `gr([?x|?l])` we can infer `gr(?x)` and `gr(?l)`.

| **Inference rule:** termination introduction (success) | 0.4% |
|---|---|

**Description:** If $T(G)$ and $S(G)$ `=>` $T(H_1$ `&` `...` `&` $H_m)$ are derivable with length $n$ then the formula

`terminates (`$G$ `&` $H_1$ `&` `...` `&` $H_m$`)`

is derivable with length $n + 1$. Here, $T$ is the built-in function that associates to a goal a formula that expresses its termination and $S$ is the corresponding function that expresses success (see p. 46).

**Formal rule:**

$$\frac{\Gamma; \Delta \rhd^n T(G) \qquad \Gamma; \Delta \rhd^n S(G)\ \texttt{=>}\ T(H_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ H_m)}{\Gamma; \Delta \rhd^{n+1} \texttt{terminates (}G\ \texttt{\&}\ H_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ H_m\texttt{)}}$$

**Example:** From

```
terminates reverse(?l1,?l2)
succeeds reverse(?l1,?l2) => terminates append(?l2,[?x],?l3)
```

we can infer `terminates (reverse(?l1,?l2) & append(?l2,[?x],?l3))`.

**See also:** Tactic `unfold` on p. 115.

| **Inference rule:** implication introduction (right) | 0.4% |
|---|---|

**Description:** If we can infer $\psi$ with length $n$ then we can infer $\varphi$ `=>` $\psi$ with length $n+1$.

**Formal rule:**

$$\frac{\Gamma; \Delta \rhd^{n} \psi}{\Gamma; \Delta \rhd^{n+1} \varphi \texttt{ => } \psi}$$

**Example:** From `q` we can infer `p => q`.

| **Inference rule:** ground introduction (term) | 0.2% |
|---|---|

**Description:** If $t$ is a pure constructor term and `gr(`$x$`)` is derivable with length $n$ for each variable $x$ of $t$, then we can derive the formula `gr(`$t$`)` with length n+1.

**Formal rule:**

$$\frac{t \in \mathit{Pure\_Terms} \quad \Gamma; \Delta \rhd^{n} \texttt{gr(}x\texttt{)} \quad \text{for all } x \in \mathrm{FV}(t)}{\Gamma; \Delta \rhd^{n+1} \texttt{gr(}t\texttt{)}}$$

**Example:** From `gr(?x)` and `gr(?l)` we can infer `gr([?x|?l])`.

| **Inference rule:** totality | 0.2% |
|---|---|

**Description:** From the formula `(terminates` $A$`)` we can infer the following formulas:

`succeeds` $A$ `\/ fails` $A$
`fails` $A$ `\/ succeeds` $A$

**Formal rule:**

$$\frac{\texttt{(terminates } A\texttt{) } \mathit{in}\ \Delta}{\Gamma; \Delta \rhd^{n} \texttt{succeeds } A \texttt{ \textbackslash/ fails } A} \qquad \frac{\texttt{(terminates } A\texttt{) } \mathit{in}\ \Delta}{\Gamma; \Delta \rhd^{n} \texttt{fails } A \texttt{ \textbackslash/ succeeds } A}$$

| **Inference rule:** termination introduction (failure) | 0.1% |
|---|---|

**Description:** If $T(G)$ and $F(G)$ are derivable with length $n$, then the formula

`terminates (`$G$ `&` $H_1$ `& ... &` $H_m$`)`

is derivable with length $n+1$. Here, $T$ is the built-in function that associates to a goal a formula that expresses its termination and $F$ is the corresponding functions that expresses its failure (see p. 46).

**Formal rule:**

$$\frac{\Gamma;\Delta \rhd^n T(G) \qquad \Gamma;\Delta \rhd^n F(G)}{\Gamma;\Delta \rhd^{n+1} \texttt{terminates (}G\ \texttt{\&}\ H_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ H_m\texttt{)}}$$

| **Inference rule:** implication introduction | 0.1% |
|---|---|

**Description:** If $\psi$ is derivable from $\Delta$ plus $\varphi$ with length $n$ then $\varphi$ `=>` $\psi$ is derivable from $\Delta$ alone with length $n+1$.

**Formal rule:**

$$\frac{\Gamma,\varphi;\Delta,\varphi \rhd^n \psi}{\Gamma;\Delta \rhd^{n+1} \texttt{(}\varphi\ \texttt{=>}\ \psi\texttt{)}}$$

| **Inference rule:** equivalence introduction | 0.1% |
|---|---|

**Description:** From $\varphi$ `=>` $\psi$ and $\psi$ `=>` $\varphi$ we can infer $\varphi$ `<=>` $\psi$.

**Formal rule:**

$$\frac{\texttt{(}\varphi\ \texttt{=>}\ \psi\texttt{)}\ in\ \Delta \qquad \texttt{(}\psi\ \texttt{=>}\ \varphi\texttt{)}\ in\ \Delta}{\Gamma;\Delta \rhd^n \texttt{(}\varphi\ \texttt{<=>}\ \psi\texttt{)}}$$

| **Inference rule:** implication introduction (left) | 0.1% |
|---|---|

**Description:** If we can infer `~`$\varphi$ with length $n$ then we can infer $\varphi$ `=>` $\psi$ with length $n+1$.

**Formal rule:**

$$\frac{\Gamma;\Delta \rhd^n \texttt{\~{}}\varphi}{\Gamma;\Delta \rhd^{n+1} \varphi\ \texttt{=>}\ \psi}$$

**Example:** From `~p` we can infer `p => q`.

| **Inference rule:** trivial equivalences | 0.1% |
|---|---|

**Description:** If $\varphi$ is in $\Delta$ and $\varphi$ is equivalent to $\psi$ then we can infer $\psi$. Equivalent means the following:

1. $s$ `<>` $t$ `<=>` `~(`$s$ `=` $t$`)`
2. $s$ `<>` $t$ `<=>` $t$ `<>` $s$

**Formal rules:**

$$\frac{s \texttt{ <> } t \; in \; \Delta}{\Gamma; \Delta \rhd^n \texttt{~(}s \texttt{ = } t\texttt{)}} \qquad \frac{\texttt{~(}s \texttt{ = } t\texttt{)} \; in \; \Delta}{\Gamma; \Delta \rhd^n s \texttt{ <> } t} \qquad \frac{s \texttt{ <> } t \; in \; \Delta}{\Gamma; \Delta \rhd^n t \texttt{ <> } s}$$

| **Inference rule:** disjunction introduction (subset) | 0.0% |
|---|---|

**Description:** A disjunction `(`$\varphi_1$ `\/ ... \/` $\varphi_n$`)` can be inferred if there exists a disjunction `(`$\psi_1$ `\/ ... \/` $\psi_m$`)` in $\Delta$ such that $\{\psi_1, \ldots, \psi_m\}$ is a subset of $\{\varphi_1, \ldots, \varphi_n\}$.

**Formal rule:**

$$\frac{\texttt{(}\psi_1 \texttt{ \/ ... \/ } \psi_m\texttt{)} \; in \; \Delta \qquad \{\psi_1, \ldots, \psi_m\} \subseteq \{\varphi_1, \ldots, \varphi_n\}}{\Gamma; \Delta \rhd^n \texttt{(}\varphi_1 \texttt{ \/ ... \/ } \varphi_n\texttt{)}}$$

**Example:** From `p \/ q` we can infer `q \/ r \/ p`.

| **Inference rule:** forall introduction (alpha conversion) | 0.0% |
|---|---|

**Description:** If $\varphi$ is in $\Delta$ and the variables $\vec{x}$ do not appear free in any formula of $\Gamma$, then we can infer `(all [`$\vec{x}$`]: `$\varphi$`)`.

**Formal rule:**

$$\frac{\varphi \; in \; \Delta \qquad \forall y \in \mathrm{FV}(\varphi)(y \in \mathrm{FV}(\Gamma) \rightarrow y\sigma \equiv y \wedge y \notin \{\vec{x}\})}{\Gamma; \Delta \rhd^n \texttt{(all [}\vec{x}\texttt{]: } \varphi\sigma\texttt{)}}$$

**Remark:** The forall introduction rule is implemented in a very general way that includes also a hidden substitution rule and still is as easy to check as the plain rule.

| **Inference rule:** equivalence elimination | 0.0% |
|---|---|

**Description:** If $\varphi$ is in $\Delta$ and $\varphi$ `<=>` $\psi$ or $\psi$ `<=>` $\varphi$ is in $\Delta$ and, then we can infer $\psi$.

**Formal rules:**

$$\frac{(\varphi \texttt{ <=> } \psi) \textit{ in } \Delta \qquad \varphi \textit{ in } \Delta}{\Gamma; \Delta \rhd^n \psi} \qquad \frac{(\psi \texttt{ <=> } \varphi) \textit{ in } \Delta \qquad \varphi \textit{ in } \Delta}{\Gamma; \Delta \rhd^n \psi}$$

| **Inference rule:** equality (injective) | 0.0% |
|---|---|

**Description:** From $f\texttt{(}s_1\texttt{,...,}s_n\texttt{)} = f\texttt{(}t_1\texttt{,...,}t_n\texttt{)}$ we can infer $s_i = t_i$ under condition that $f$ is a constructor symbol and not a defined function symbol.

**Formal rule:**

$$\frac{f\texttt{(}s_1\texttt{,...,}s_n\texttt{)} = f\texttt{(}t_1\texttt{,...,}t_n\texttt{)} \textit{ in } \Delta \qquad f \in \textit{Constructors}}{\Gamma; \Delta \rhd^n s_i = t_i}$$

**Example:** From `[?x|?l1] = [?y|?l2]` we can infer `?x = ?y` and `?l1 = ?l2`.

| **Inference rule:** alpha conversion | 0.0% |
|---|---|

**Description:** If $\psi$ is in $\Delta$ and $\varphi$ is obtained from $\psi$ by renaming bound variables, then we can infer $\varphi$.

**Formal rule:**

$$\frac{\psi \textit{ in } \Delta \qquad \psi \equiv \varphi}{\varphi}$$

Remember, that we write $\varphi \equiv \psi$ to express that $\varphi$ is alpha equivalent to $\psi$.

**Remark:** The renaming of bound variables is included into the matching primitives. So the rule for alpha conversion is not used very often.

## 3.4 Inference rules with names

In this section we define the relation $\Gamma\{\varphi \texttt{ by } tag\}\Delta \rightsquigarrow \psi$ for various different kinds of *tags*. In all cases except in the case that $\varphi$ is of the form `(def` $\chi$`)` the formula $\psi$ that is introduced by the derivation step is the formula $\varphi$ itself. Remember that derivation steps of the form `(`$\varphi$ `by` *tag*`)` are used in 38.4% cases of all derivation steps. This means that they are the most used derivation steps.

The reason that there is the keyword `by` is that if would cost too much the LPTP system to figure out itself by which rule the formula has been inferred. Proof-checking would be too slow.

This section does not define all cases of *by* derivation steps. Some of them that have to do with the extension of the language by abbreviations and functions symbols and the use of built-in predicates are described in Chapter 4. The percent numbers in this section refer to the frequency of the usage of the single tags.

| **By:** reference | 51.8% |
|---|---|

**Syntax:**

*formula* `by theorem(`*reference*`)`
*formula* `by lemma(`*reference*`)`
*formula* `by corollary(`*reference*`)`
*formula* `by axiom(`*reference*`)`

**Description:** The tags `theorem`, `lemma`, `corollary` and `axiom` have the same meaning. If *fact*`(`*reference*`)` refers to the formula $\psi$ and the formula $\varphi$ is immediately derivable from $\Delta$ plus $\psi$ under the assumptions $\Gamma$, then the derivation step

$\varphi$ `by` *fact*`(`*reference*`)`

introduces the formula $\varphi$. It is assumed that $\psi$ does not contain free variables. What LPTP does, is that it looks its internal database for the *fact* $\psi$ with the name *reference*. Then it adds it temporarily to the already derived formulas.

**Formal rule:**

$$\frac{\Gamma; \Delta, \psi \triangleright \varphi}{\Gamma\{\varphi \texttt{ by } \textit{fact}\texttt{(}\textit{reference}\texttt{)}\}\Delta \rightsquigarrow \varphi}$$

**Remark:** The *reference* has to be a colon separated list of names of the form:

*name*`:`*name*`: ... :`*name*

At the moment, LPTP does not yet use the structure of references. However, in the future it is planned to used names in heuristics.

**Example:** Assume that the `corollary(lh:cons)` refers to the formula

```
all [x,l]:succeeds list(?l) => lh([?x|?l]) = s(lh(?l))
```

Then we can write:

```
assume(succeeds list(?l),
 lh([?x|?l]) = s(lh(?l)) by corollary(lh:cons),
 ...)
```

**See also:** Tactic `fact`, p. 112.

**By:** completion (fixed point) 18.0%

**Syntax:**

```
def succeeds atom by completion
def fails atom by completion
def terminates atom by completion
```

**Description:** The fixed point rule says that if the formula `(succeeds` $A$`)` is immediately derivable then the derivation step

```
def succeeds atom by completion
```

introduces the defining formula for `(succeeds` $A$`)`. The defining formula for `(succeeds` $A$`)` can be obtained with the LPTP following command (see p. 48):

```
?- def(succeeds A).
```

The same rule applies for `(fails` $A$`)` and `(terminates` $A$`)`. Note that defining formulas depend on the logic program $P$, i.e. on the clauses in the internal database of LPTP.

**Formal rules:**

$$\frac{\Gamma;\Delta \triangleright \texttt{succeeds } A}{\Gamma\{\texttt{def succeeds } A \texttt{ by completion}\}\Delta \rightsquigarrow D^P(\texttt{succeeds } A)}$$

$$\frac{\Gamma;\Delta \triangleright \texttt{fails } A}{\Gamma\{\texttt{def fails } A \texttt{ by completion}\}\Delta \rightsquigarrow D^P(\texttt{fails } A)}$$

$$\frac{\Gamma;\Delta \triangleright \texttt{terminates } A}{\Gamma\{\texttt{def terminates } A \texttt{ by completion}\}\Delta \rightsquigarrow D^P(\texttt{terminates } A)}$$

**See also:** For the defintion of $D^P$ see p. 48.

| **By:** sld | 9.7% |
|---|---|

**Syntax:**

`succeeds` *atom* `by sld`

**Description:** If there exists a clause $B$ `:-` $G$ in the internal database and a substitution $\sigma$ such that $A \equiv B\sigma$ and the success of the body $G\sigma$ follows immediately from $\Delta$, then the derivation step

`succeeds` $A$ `by sld`

derives the formula `(succeeds` $A$`)`. Now, we have to say what we mean by the success of the body $G\sigma$ follows immediately from $\Delta$. This means that the success formula $S(G\sigma)$ is immediately derivable from $\Delta$.

**Remark:** This rule is an efficient version of the rule

`succeeds` $A$ `by completion.`

**Formal rules:**

$$\frac{(A \texttt{ :- } G) \in \mathit{Clauses} \qquad \mathit{body}(\Delta, S(G\sigma))}{\Gamma\{\texttt{succeeds } A\sigma \texttt{ by sld}\}\Delta \rightsquigarrow \texttt{succeeds } A\sigma}$$

$$\frac{(t_1 \texttt{ = } t_2) \mathit{ in } \Delta}{\mathit{body}(\Delta, t_1 \texttt{ = } t_2)} \qquad \frac{(t_2 \texttt{ = } t_1) \mathit{ in } \Delta}{\mathit{body}(\Delta, t_1 \texttt{ = } t_2)}$$

$$\frac{(t_1 \texttt{ <> } t_2) \mathit{ in } \Delta}{\mathit{body}(\Delta, t_1 \texttt{ <> } t_2)} \qquad \frac{(t_2 \texttt{ <> } t_1) \mathit{ in } \Delta}{\mathit{body}(\Delta, t_1 \texttt{ <> } t_2)}$$

$$\frac{(\texttt{succeeds } A) \mathit{ in } \Delta}{\mathit{body}(\Delta, \texttt{succeeds } A)} \qquad \frac{(\texttt{fails } B) \mathit{ in } \Delta}{\mathit{body}(\Delta, \texttt{fails } A)}$$

$$\frac{\mathit{body}(\Delta, \varphi_1) \quad \ldots \quad \mathit{body}(\Delta, \varphi_n)}{\mathit{body}(\Delta, \varphi_1 \texttt{ \& ... \& } \varphi_n)} \qquad \frac{\mathit{body}(\Delta, \varphi_i)}{\mathit{body}(\Delta, \varphi_1 \texttt{ \textbackslash/ ... \textbackslash/ } \varphi_n)}$$

**See also:** See p. 46 for the definition of the formula $S(G)$.

| **By:** completion (closure) | 5.0% |
|---|---|

**Syntax:**

```
succeeds atom by completion
fails atom by completion
terminates atom by completion
```

**Description:** If the success formula $D^P(\texttt{succeeds}\ A)$ is immediately derivable from $\Delta$, then the derivation step

```
succeeds A by completion
```

derives the formula `(succeeds A)`.

If the failure formula $D^P(\texttt{fails}\ A)$ is immediately derivable from $\Delta$, then the derivation step

```
fails A by completion
```

derives the formula `(fails A)`.

If the termination formula $D^P(\texttt{fails}\ A)$ is immediately derivable from $\Delta$, then the derivation step

```
terminates A by completion
```

derives the formula `(terminates A)`.

**Formal rules:**

$$\frac{\Gamma; \Delta \triangleright D^P(\texttt{succeeds}\ A)}{\Gamma\{\texttt{succeeds}\ A\ \texttt{by completion}\}\Delta \rightsquigarrow \texttt{succeeds}\ A}$$

$$\frac{\Gamma; \Delta \triangleright D^P(\texttt{fails}\ A)}{\Gamma\{\texttt{fails}\ A\ \texttt{by completion}\}\Delta \rightsquigarrow \texttt{fails}\ A}$$

$$\frac{\Gamma; \Delta \triangleright D^P(\texttt{terminates}\ A)}{\Gamma\{\texttt{terminates}\ A\ \texttt{by completion}\}\Delta \rightsquigarrow \texttt{terminates}\ A}$$

**See also:** For the defintion of $D^P$ see p. 48.

**By:** gap

**Syntax:**

*formula* `by gap`

**Description:** A derivation can contain gaps. Of course, as long as a derivation contains gaps it is not a correct derivation in the logical sense. The pseudo derivation step $\varphi$ `by gap` derives the formula $\varphi$. In other words, it forces the formula $\varphi$ to be true. The LPTP system prints a warning if a derivation contains gaps.

**Formal rule:**

$$\overline{\Gamma\{\varphi \texttt{ by gap}\}\Delta \rightsquigarrow \varphi}$$

**Example:**

```
ff by gap
```

**By:** because

**Syntax:**

*formula* `by because`

**Descritption:** The `because` derivation step is exactly as the `gap` derivation step except that LPTP does not print a warning message.

**Formal rule:**

$$\overline{\Gamma\{\varphi \texttt{ by because}\}\Delta \rightsquigarrow \varphi}$$

## Derivations containing gaps

The last two derivation steps are pseudo derivation steps, since they allow the user to introduce arbitrary formulas. For example, even the constant `ff` (falsum) can be introduced in this way. Although the proofs may contain gaps they are still partially correct. This means that formulas that are not marked by `gap` or `because` still have to be derivable from other formulas according to the inference rules. Moreover, the gaps provied the user a means to construct proofs in an arbitrary way: from assumptions forward to the conclusion (forward chaining) or from the conclusion backward to the assumptions (backward chaining). It allows even arbitrary combinations of both styles.

*A rule of inference is nothing else but a mechanical procedure which allows one to determine of any given finite class of expressions whether anything can be inferred from them by means of the rule of inference under consideration, and if so to write down the conclusion. (. . . ) Now you see the essential difficulty of this definition is the notion of mechanical procedure which comes in and which needs further specification. These mechanical procedures are here applied to expressions or rather finite classes of expressions, and the result is either again an expression or the answer Yes or No.*

*K. Gödel, 193?*
*"Collected Works, Volume III, p. 166"*

# Chapter 4

# More about formal proofs

At this point we have to make some thoughts about the relation of formal proofs and the reality. For example, why does a formal proof for the equivalence of two predicates imply that the two predicates are computationally equivalent in reality? Why does a formal proof of termination of a predicate imply that the predicate terminates in reality? What does reality mean here? What is a formal proof more than a string of 0's and 1's in the memory of a computer?

We cannot answer all the questions here because some of them are philosophical questions about the foundations of mathematics and do not have answers. However, we can explain what the assumptions are that we make about the world and why under these assumptions it follows that the LPTP system or at least the underlying theory is adequate for its purpose. For the moment, let us assume that there are no bugs in LPTP and that it implements exactly the formal system described in [16] which is an extension of the formal system of [11].

The main assumption about Prolog that we make is that it implements correctly the mathematical operational model of [16] and, for example, performs the occurs-check during unification if it is necessary. If we do not make this assumption, then the LPTP system is not correct for Prolog. For example, we can easily derive in LPTP the following formula:

```
terminates (?x = f(?x) & ?y = f(?y) & ?x = ?y).
```

The corresponding Prolog query, however, does not terminate in standard implementations of Prolog, since they omit the occurs check.

```
?- X = f(X), Y = f(Y), X = Y.
```

We consider it as a definitive mistake in the design of Prolog that the above query does not terminate.

The second assumptions about Prolog is that it also correctly implements the built-in predicates according to the model of [16]. We will say more about built-in predicates in Section 4.3 of this chapter. For the moment we just want to mention that our model for built-in predicates is simple and flexible and

comprises most of the so-called logical built-in predicates of Prolog and even non-logical predicates like `call/1`. However, it is not possible to treat predicates like the `var/1` predicate. The reason is simple. Under any reasonable axioms for `var/1` it must certainly be possible do derive `succeeds var(?x)`, since `?x` is a variable. But then we can derive:

```
all x: succeeds var(?x)
succeeds var(c)
```

But `c` is a constant and `var(c)` fails. Thus the LPTP system would be incorrect. We believe that the era of `var/1` is over (cf [9]).

Under the two assumptions that the occurs check is done and that the built-in predicates are implemented correctly, one can show that LPTP is adequate in the following sense. Let us assume that $P$ is a pure Prolog program and that the internal database of LPTP contains exactly the clauses of $P$ and nothing else. We write $P \vdash \varphi$, if there exists a derivation $d$ such that LPTP accepts $d$ as a correct derivation for $\varphi$ in the sense of page 3.1. Then the results of [16] say the following:

(T1) If $A$ has answer $\sigma$ in Prolog, then $P \vdash$ `succeeds` $A\sigma$.

(T2) If $A$ has answer `no` in Prolog, then $P \vdash$ `fails` $A$.

These are the simple results. The non-trivial results say the following:

(T3) If $P \vdash$ `gr(`$t$`)`, then $t$ is a ground term.

(T4) If $P \vdash$ `terminates` $A$, then the Prolog computation for $A$ terminates. This means that $A$ has finitely many answers, i.e., one can hit the semicolon key a finite number of times until one finally gets the answer `no`. It also means that during the computation, negative goals are always ground if they are called. Moreover, built-in atoms are correctly instantiated such that they can be evaluated when they are called. In fact, it means even more. One can permute the clauses in the program $P$ in an arbitrary way (even during run-time) and the computation still terminates and does not flounder.

(T5) If $P \vdash$ `terminates` $A$ `& (ex [`$\vec{x}$`]: succeeds` $A$`)`, then there exists an answer for $A$ (not necessarily the first) that has answer terms for the variables $\vec{x}$ only.

(T6) If $P \vdash$ `terminates` $A$ `& fails` $A$, then $A$ has answer `no` in Prolog.

(T7) If $P \vdash$ `terminates` $A$ `& succeeds` $A\sigma$, then one of the answers of Prolog is more general than $\sigma$.

What we see from (T5)–(T7) is that we always have to prove termination, otherwise we cannot apply the formal statements to Prolog. It is easy to give an example, why it is really necessary. Consider the program $P$ that has the following clauses:

```
r :- q, fail.
q :- q.
```

Then the following is a correct LPTP derivation for `fails r`.

```
:- lemma(loop,
 fails r,
 [fails fail, fails q \/ fails fail, fails r]
).
```

However, the goal `?- r` does not fail in Prolog. It loops. This is not a contradiction to (T6), since we did not prove `terminates r` and, in fact, this is not possible in LPTP.

Hence, we have always to prove termination of the predicates. This may seem cumbersome, since if we have proved termination, then we have proved too much. We have shown that the order of clauses in Program is irrelevant which is much more than just termination under Prolog. However, in practice this is not a problem as has been observed by Apt and Pedreschi in [1, 2]. — Our examples in Chapter 7 confirm their thesis.

## 4.1 Defining predicates

In LPTP the user has the possibility to define abbreviations. Formally, an abbreviation is an extension of the language. It is well-know that under certain assumption the extension of a formal language by new predicates is conservative over the original language. This means that if we have a derivation of a formula that uses abbreviations and the formula itself does not contain abbreviations, then the derivation can be expanded into a derivation of the original language that does not contain abbreviations.

The principle of language extensions is true for pure predicate logic. As soon as non-logical principles are involved as for example, induction or equality axioms, one has to make sure that the conservativeness property remains true. In our case this is not a problem.

The best way to explain abbreviations is by looking at an example. We recall the predicate definition of the introduction:

```
:- definition_pred(sub,2,
all [l1,l2]: sub(?l1,?l2) <=>
 (all x: succeeds member(?x,?l1) => succeeds member(?x,?l2))
).
```

This defintion defines a binary predicate `sub` that expresses the subset relation between lists. In general, a predicate definition has the following form:

```
:- definition_pred(R,n,all [x1,...,xn]: R(x1,...,xn) <=> φ)
```

where the free variables of $\varphi$ are contained in $x_1, \ldots, x_n$.

When LPTP processes a predicate definition it first checks whether the symbol $R$ is already defined as a predicate or as a function symbol. If this is the case, then it prints an error message. Otherwise the definition is added to the internal database and also written on the '`.thm`' file. The latter is important, since if the theorems are later used than one has to ensure that the abbreviations in the theorems have their original meaning.

As soon as LPTP has a definition for a symbol it treats the symbol differently. In our example, it changes the internal representation of `sub` and it is no longer possible to use `sub` in another context, say as constructor symbol or predicate symbol in a logic program.

Although LPTP does only enforce it as far as it is really necessary for the soundness of the system, the following three name spaces should be disjoint:

1. names used in logic programs (clauses),
2. names used in predicate definitions,
3. names used function definitions.

The inference rules are implemented such that it is not possible to use a name in different contexts at the same time. For example, as soon as `sub` has a predicate definition the axiom

```
sub(?x,?y) = sub(?u,?v) => ?x = ?y & ?y = ?v
```

can no longer be applied, since `sub` is no longer a constructor symbol.

There are two inference rules associated to a predicate definition: an introduction rule and an elimination rule. Introduction means replacing the formula by the abbreviation and elimination means expanding an abbreviation into its definition.

| **By:** predicate introduction | 4.0% |
|---|---|

**Syntax:**

*name*`(`*term*$_1$`,...,`*term*$_n$`) by introduction(`*name*`,`$n$`)`

**Description:** Assume that $R$ is defined by

```
:- definition_pred(R,n,
all [x_1,...,x_n]: R(x_1,...,x_n) <=> φ(x_1,...,x_n)
).
```

Then, if $\varphi(t_1, \ldots, t_n)$ is immediately derivable from $\Delta$, the derivation step

$R$`(`$t_1$`,...,`$t_n$`) by introduction(`$R$`,`$n$`)`

derives the formula $R(t_1, \ldots, t_n)$.

**Formal rule:**

$$\frac{\Gamma; \Delta \triangleright \varphi(t_1, \ldots, t_n) \qquad \texttt{(all [}\vec{x}\texttt{]: } R\texttt{(}\vec{x}\texttt{) <=> } \varphi(\vec{x})\texttt{)} \in \mathit{Def}}{\Gamma\{R\texttt{(}t_1\texttt{,...,}t_n\texttt{) by introduction(}R\texttt{,}n\texttt{)}\}\Delta \rightsquigarrow R(t_1, \ldots, t_n)}$$

**Example:** Assume that `?x` is not free in $\Gamma$ and that `sub` is defined as at the beginning of this section. Then the following is a correct derivation:

```
[succeeds member(?x,?l1) => succeeds member(?x,?l2) by gap,
 sub(?l1,?l2) by introduction(sub,2)]
```

**See also:** Tactic `unfold` on p. 115.

| **By:** predicate elimination | 4.4% |
|---|---|

**Syntax:**

*formula* `by elimination(`*name*`,`*arity*`)`

**Description:** Assume that $R$ is defined by

`:- definition_pred(`$R$`,`$n$`,`
`all [`$x_1$`,...,`$x_n$`]:` $R$`(`$x_1$`,...,`$x_n$`)` `<=>` $\varphi(x_1, \ldots, x_n)$
`).`

Then, if $R(t_1, \ldots, t_n)$ is immediately derivable from $\Delta$, the derivation step

$\varphi$`(`$t_1$`,...,`$t_n$`)` `by elimination(`$R$`,`$n$`)`

derives the formula $\varphi(t_1, \ldots, t_n)$.

**Formal rule:**

$$\frac{\Gamma; \Delta \triangleright R(t_1, \ldots, t_n) \qquad \texttt{(all [}\vec{x}\texttt{]: } R(\vec{x}) \texttt{ <=> } \varphi(\vec{x})\texttt{)} \in \mathit{Def}}{\Gamma\{\varphi(t_1, \ldots, t_n) \texttt{ by elimination(}R\texttt{,}n\texttt{)}\}\Delta \rightsquigarrow R(t_1, \ldots, t_n)}$$

**Example:** Assume that `sub` is defined as at the beginning of this section. Then the following derivation is correct:

```
[sub(?l1,?l2) by gap,
 member(?x,?l1) by gap,
 all x: succeeds member(?x,?l1) => succeeds member(?x,?l2)
  by elimination(sub,2),
 member(?x,?l2)]
```

**See also:** Tactic `def` on p. 110.

## 4.2 Defining function symbols

Language extensions by function symbols are not that simple as extensions by predicate symbols. Defined function symbols are not just abbreviations. Defined functions symbols replace existential quantifiers and that is the reason why they are useful. Function definitions have the following form:

```
:- definition_fun(f,n,
 all [x1,...,xn,y]: δ(x1,...,xn) =>
  (f(x1,...,xn) = y <=> γ(x1,...,xn,y)),
 existence by fact(reference1),
 uniqueness by fact(reference2)
).
```

The name $f$ is the function symbol that is defined, $n$ is the number of its arguments, $\delta$ is the domain of the function and $\gamma$ is the graph of the function. The definition is only accepted by LPTP if *reference*$_1$ refers to

```
all [x1,...,xn]: δ(x1,...,xn) => (ex y: γ(x1,...,xn,y))
```

and *reference*$_2$ to

```
all [x1,...,xn,y,z]: δ(x1,...,xn) & γ(x1,...,xn,y) &
 γ(x1,...,xn,z) => y = z
```

or to facts from which the two formulas are immediately derivable. When LPTP processes a function definitions it first checks the syntactic correcteness and then tries to derive the existence statement and the uniqueness statement from the corresponding facts. It it succeeds then the function definition is added to the internal database and also written on the '`.thm`' file.

Let us consider an example about the concatenation of lists. The predicate `append/3` from p. 8 has the following two properties:

```
lemma(append:existence,
 all [l1,l2]: succeeds list(?l1) =>
  (ex l3: succeeds append(?l1,?l2,?l3)),
 derivation
).

lemma(append:uniqueness,
 all [l1,l2,l3,l4]: succeeds append(?l1,?l2,?l3) &
  succeeds append(?l1,?l2,?l4) => ?l3 = ?l4,
 derivation
).
```

The two properties can be found in the library file `$(lib)/list/list.pr`. And as soon as they are in the internal database of LPTP we can define a binary function symbol `**` in the following way:

```
:- definition_fun(**,2,
 all [l1,l2,l3]: succeeds list(?l1) =>
  (?l1 ** ?l2 = ?l3 <=> succeeds append(?l1,?l2,?l3)),
 existence by lemma(append:existence),
 uniqueness by lemma(append:uniqueness)
).
```

In this definition, the domain and graph of of `**/2` are the following formulas:

$\delta$(?l1,?l2) := `succeeds list(?l2)`
$\gamma$(?l1,?l2,?l3) := `succeeds append(?l1,?l2,?l3)`

Note that `**/2` is treated as a total function. If the second argument of `**/2`, however, is not a list, or if it is not provably a list, then we know nothing about the value of `?l1 ** ?l2`. Note also, that LPTP treats `**/2` no longer as a constructor symbol. Thus, for example, the axioms

```
?l1 ** ?l2 <> 0
?l1 ** ?l2 = ?m1 ** ?m2 => ?l1 = ?m1 & ?l2 = ?m2
```

are no longer available.

There are two inference rules associated to a function definiton: the existence and the uniqueness rule.

| **By:** function existence | 2.6% |
|---|---|

**Syntax:**

*formula* `by existence(`*name*`,`*arity*`)`

**Description:** Assume that $f$ has the following definition:

```
:- definition_fun(f,n,
 all [x⃗,y]: δ(x⃗) => (f(x⃗) = y <=> γ(x⃗,y)),
 existence by fact(reference_1),
 uniqueness by fact(reference_2)
).
```

Then, if $\delta(\vec{t})$ is immediately derivable from $\Delta$, the derivation step

$\gamma(\vec{t}, f(\vec{t}))$ `by existence(`$f$`,`$n$`)`

derives the formula $\gamma(\vec{t}, f(\vec{t}))$.

**Formal rule:**

$$\frac{\Gamma; \Delta \triangleright \delta(\vec{t}) \qquad \texttt{(all [}\vec{x},y\texttt{]: }\delta(\vec{x})\texttt{ => (}f\texttt{(}\vec{x}\texttt{) = }y\texttt{ <=> }\gamma(\vec{x},y)\texttt{))} \in \mathit{Def}}{\Gamma\{\gamma(\vec{t}, f(\vec{t}))\texttt{ by existence(}f\texttt{,}n\texttt{)}\}\Delta \rightsquigarrow \gamma(\vec{t}, f(\vec{t}))}$$

**Example:** Assume that `**/2` has the standard definition from above. Then the following is a correct derivation:

```
[succeeds list(?l2) by gap,
 succeeds append(?l1,?l2,?l1 ** ?l2) by existence(**,2)]
```

| **By:** function uniqueness | 1.8% |
|---|---|

**Syntax:**

*term* `=` *term* `by uniqueness(`*name*`,`*arity*`)`

**Description:** Assume that $f$ has the following definition:

```
:- definition_fun(f,n,
 all [x⃗,y]: δ(x⃗) => (f(x⃗) = y <=> γ(x⃗,y)),
 existence by fact(reference_1),
 uniqueness by fact(reference_2)
).
```

Then, if $\delta(\vec{t})$ and $\gamma(\vec{t},s)$ are immediately derivable from $\Delta$, the derivation step

$f(\vec{t})$ `=` $s$ `by uniqueness(`$f$`,`$n$`)`

derives the formula $f(\vec{t})$ `=` $s$ and the derivation step

$s$ `=` $f(\vec{t})$ `by uniqueness(`$f$`,`$n$`)`

derives the formula $s$ `=` $f(\vec{t})$.

**Formal rules:**

$$\frac{\Gamma;\Delta \triangleright \delta(\vec{t}) \qquad \Gamma;\Delta \triangleright \gamma(\vec{t},s) \qquad \texttt{(all [}\vec{x},y\texttt{]: }\delta(\vec{x})\texttt{ => (}f\texttt{(}\vec{x}\texttt{) = }y\texttt{ <=> }\gamma(\vec{x},y)\texttt{))} \in \mathit{Def}}{\Gamma\{f(\vec{t})\texttt{ = }s\texttt{ by uniqueness(}f\texttt{,}n\texttt{)}\}\Delta \rightsquigarrow f(\vec{t})\texttt{ = }s}$$

$$\frac{\Gamma;\Delta \triangleright \delta(\vec{t}) \qquad \Gamma;\Delta \triangleright \gamma(\vec{t},s) \qquad \texttt{(all [}\vec{x},y\texttt{]: }\delta(\vec{x})\texttt{ => (}f\texttt{(}\vec{x}\texttt{) = }y\texttt{ <=> }\gamma(\vec{x},y)\texttt{))} \in \mathit{Def}}{\Gamma\{s\texttt{ = }f(\vec{t})\texttt{ by uniqueness(}f\texttt{,}n\texttt{)}\}\Delta \rightsquigarrow s\texttt{ = }f(\vec{t})}$$

**Example:** Assume that `**/2` has the standard definition from above. Then the following is a correct derivation:

```
[succeeds list(?l2) by gap,
 succeeds append(?l1,?l2,[?x|?l3]) by gap,
 ?l1 ** ?l2 = [?x|?l3] by uniqueness(**,2),
 [?x|?l3] = ?l1 ** ?l2 by uniqueness(**,2)]
```

## 4.3 Axioms and built-in predicates

Most Prolog programs use built-in predicates. Therefore it is important for a system like LPTP to support built-in predicates. The addition of built-in predicates to LPTP, however, is not so trivial as it may seem. The fundamental questions are: What are the right axioms for the built-in predicates? Why do axioms like

```
all [x,y]: succeeds ?x =< ?y => ?x = ?y \/ succeeds ?x < ?y,
all x: succeeds call(list,?x) <=> succeeds list(?x),
all [x1,x2,y]: succeeds ?x1 is ?y & succeeds ?x2 is ?y =>
 ?x1 = ?x2
```

not destroy the properties (T1)–(T7) of LPTP? — The answers are given in [16]. Here, we just recall what is necessary to use the LPTP system.

An atom $R(t_1,\ldots,t_n)$ is called a *built-in* atom, if $R$ is a built-in predicate. The built-in in predicates that LPTP supports at the moment are:

```
atom/1
integer/1
atomic/1
is/2
</2
=</1
call/n + 1
current_op/3
```

This list can be extended by arbitrary built-in predicates provided that they they fit into the following model.

### Evaluation of built-in predicates

We assume that the result of the evaluation of a built-in atom is a goal. We write $\mathit{eval}(A)$ for the result of the evaluation of $A$. Thus $\mathit{eval}(A)$ is a goal. Since not every built-in atom can be evaluated, the function *eval* is a partial function. We assume that *eval* has the following two properties:

(D) If $A \in \mathrm{dom}(\mathit{eval})$, then $A\sigma \in \mathrm{dom}(\mathit{eval})$ for each substitution $\sigma$.

(E) $\mathit{eval}(A\sigma) \equiv \mathit{eval}(A)\sigma$ for all $A \in \mathrm{dom}(\mathit{eval})$ and all substitutions $\sigma$.

Condition (D) says that if an atom can be evaluated then every instance of the atom can also be evaluated. Condition (E) says that the evaluation of an instance of an atom must be compatible with the evaluation of the atom itself. This condition is needed to make the *lifting lemma* true.

The function *eval* is defined in the following way:

**Built-in predicate:** `atom/1`

**Type condition:** `atom(`$t$`)` $\in \mathrm{dom}(eval) :\Longleftrightarrow$ $t$ is a ground term.

**Evaluation:**

$$eval(\texttt{atom(}t\texttt{)}) := \begin{cases} \texttt{true}, & \text{if } t \text{ is a constant but not a number;} \\ \texttt{fail}, & \text{otherwise.} \end{cases}$$

**Example:** $eval(\texttt{atom(c)}) = \texttt{true}$, $eval(\texttt{atom(f(0))}) = \texttt{fail}$, $eval(\texttt{atom(2)}) = \texttt{fail}$.

**Built-in predicate:** `integer/1`

**Type condition:** `integer(`$t$`)` $\in \mathrm{dom}(eval) :\Longleftrightarrow$ $t$ is a ground term.

**Evaluation:**

$$eval(\texttt{integer(}t\texttt{)}) := \begin{cases} \texttt{true}, & \text{if } t \text{ is an integer;} \\ \texttt{fail}, & \text{otherwise.} \end{cases}$$

**Example:** $eval(\texttt{integer(2)}) = \texttt{true}$, $eval(\texttt{integer(c)}) = \texttt{fail}$.

**Built-in predicate:** `atomic/1`

**Type condition:** `atomic(`$t$`)` $\in \mathrm{dom}(eval) :\Longleftrightarrow$ $t$ is a ground term.

**Evaluation:**

$$eval(\texttt{atomic(}t\texttt{)}) := \begin{cases} \texttt{true}, & \text{if } t \text{ is a constant or a number;} \\ \texttt{fail}, & \text{otherwise.} \end{cases}$$

**Example:** $eval(\texttt{atomic(2)}) = \texttt{true}$, $eval(\texttt{atomic(c)}) = \texttt{true}$, $eval(\texttt{atomic(f(c))}) = \texttt{fail}$.

**Built-in predicate:** `is/2`

**Type condition:** `(`$s$ `is` $t$`)` $\in \mathrm{dom}(eval) :\Longleftrightarrow$ $t$ is a ground arithmetic term, i.e. a term built-up from integer constants `0`, `1`, `-1`, `2`, `-2`, ... using the arithmetical operators `+`, `-`, `*`, `/`.

**Evaluation:**

$eval(\texttt{(}s$ `is` $t\texttt{)}) := \texttt{(}s$ `=` $n\texttt{)}$, where $n$ is the value of the term $t$.

**Example:** $eval(\texttt{(?x is (2 * 2) + 3)}) := \texttt{(?x = 7)}$.

**Built-in predicate:** `</2`

**Type condition:** `(`$s$ `<` $t$`)` $\in \mathrm{dom}(eval) :\Longleftrightarrow$ $s$ and $t$ are ground arithmetic terms (see `is/2`).

**Evaluation:**

$$eval(s \texttt{ < } t) := \begin{cases} \texttt{true}, & \text{if the value of } s \text{ is less than the value of } t; \\ \texttt{fail}, & \text{otherwise.} \end{cases}$$

**Example:** $eval(\texttt{2 < 3}) = \texttt{true}$, $eval(\texttt{2 < 2}) = \texttt{fail}$, $eval(\texttt{3 < 2}) = \texttt{fail}$.

**Built-in predicate:** `=</2`

**Type condition:** $(s \texttt{ =< } t) \in \mathrm{dom}(eval) :\Longleftrightarrow s$ and $t$ are ground arithmetic terms (see `is/2`).

**Evaluation:**

$$eval(s \texttt{ =< } t) := \begin{cases} \texttt{true}, & \text{if the value of } s \text{ is less than or equal to} \\ & \text{the value of } t; \\ \texttt{fail}, & \text{otherwise.} \end{cases}$$

**Example:** $eval(\texttt{2 =< 3}) = \texttt{true}$, $eval(\texttt{2 =< 2}) = \texttt{true}$, $eval(\texttt{3 =< 2}) = \texttt{fail}$.

**Built-in predicate:** `call/`$n+1$

**Type condition:** $\texttt{call(}s\texttt{,}t_1\texttt{,...,}t_n\texttt{)} \in \mathrm{dom}(eval) :\Longleftrightarrow s$ is a constant.

**Evaluation:**

$eval(\texttt{call(}s\texttt{,}t_1\texttt{,...,}t_n\texttt{)}) := s\texttt{(}t_1\texttt{,...,}t_n\texttt{)}$.

**Example:** $eval(\texttt{call(list,[1,2])}) = \texttt{list([1,2])}$,
$eval(\texttt{call(<,?x,?y)}) = \texttt{(?x < ?y)}$.

**Built-in predicate:** `current_op/3`

**Type condition:** $\texttt{current\_op(}t_1\texttt{,}t_2\texttt{,}t_3\texttt{)} \in \mathrm{dom}(eval) :\Longleftrightarrow t_1$ is an integer, $t_2$ is one of the constants `fx`, `fy`, `xfy`, `xfx`, `yfx`, `xf`, `yf` and $t_3$ is ground.

**Evaluation:** $eval(\texttt{current\_op(}t_1\texttt{,}t_2\texttt{,}t_3\texttt{)}) := G_1$ `\/ ... \/` $G_n$,
where the goals $G_i$ correspond to the entries in the operator table, for example,

$G_i = \texttt{(}t_1\texttt{ = 500 \& }t_2\texttt{ = yfx \& }t_3\texttt{ = (+))}$.

## Rules for built-in predicates

There are two rules for built-in predicates. The derivation steps have the following form:

$\varphi$ `by builtin`

These derivation steps correspond to the `by completion` steps for user-defined predicates.

| **By:** built-in (closure) | 0.9% |
|---|---|

**Syntax:**

```
succeeds atom by builtin
fails atom by builtin
terminates atom by builtin
```

**Description:** Assume that $A$ is a built-in atom such that $A \in \mathrm{dom}(\mathit{eval})$ and $\mathit{eval}(A) = G$. If $S(G)$ is immediately derivable from $\Delta$ then the derivation step

```
succeeds A by builtin
```

derives the formula `(succeeds A)`. If $F(G)$ is immediately derivable from $\Delta$ then the derivation step

```
fails A by builtin
```

derives the formula `(fails A)`. If $T(G)$ is immediately derivable from $\Delta$ then the derivation step

```
terminates A by builtin
```

derives the formula `(terminates A)`.

**Formal rules:**

$$\frac{A \in \mathrm{dom}(\mathit{eval}) \qquad \Gamma; \Delta \triangleright S(\mathit{eval}(A))}{\Gamma\{\texttt{succeeds } A \texttt{ by builtin}\}\Delta \rightsquigarrow \texttt{succeeds } A}$$

$$\frac{A \in \mathrm{dom}(\mathit{eval}) \qquad \Gamma; \Delta \triangleright F(\mathit{eval}(A))}{\Gamma\{\texttt{fails } A \texttt{ by builtin}\}\Delta \rightsquigarrow \texttt{fails } A}$$

$$\frac{A \in \mathrm{dom}(\mathit{eval}) \qquad \Gamma; \Delta \triangleright T(\mathit{eval}(A))}{\Gamma\{\texttt{terminates } A \texttt{ by builtin}\}\Delta \rightsquigarrow \texttt{terminates } A}$$

**Example:** The following is a correct derivation:

```
[succeeds 0 < 1 by builtin,
 terminates ?x < ?y by gap,
 terminates call(<,?x,?y) by builtin]
```

**See also:** The functions $S$, $F$ and $T$ are defined on page 46.

| **By:** built-in (fixed point) | 0.3% |
|---|---|

**Syntax:**

```
def succeeds atom by builtin
def fails atom by builtin
def terminates atom by builtin
```

**Description:** Assume that $A$ is a built-in atom such that $A \in \mathrm{dom}(\mathit{eval})$ and $\mathit{eval}(A) = G$. If `(succeeds A)` is immediately derivable from $\Delta$, then the derivation step

```
def succeeds A by builtin
```

derives the formula $S(G)$. If `(fails A)` is immediately derivable from $\Delta$, then the derivation step

```
def fails A by builtin
```

derives the formula $F(G)$. If `(terminates A)` is immediately derivable from $\Delta$, then the derivation step

```
def terminates A by builtin
```

derives the formula $T(G)$.

**Formal rules:**

$$\frac{A \in \mathrm{dom}(\mathit{eval}) \qquad \Gamma; \Delta \triangleright \texttt{succeeds } A}{\Gamma\{\texttt{def succeeds } A \texttt{ by builtin}\}\Delta \rightsquigarrow S(\mathit{eval}(A))}$$

$$\frac{A \in \mathrm{dom}(\mathit{eval}) \qquad \Gamma; \Delta \triangleright \texttt{fails } A}{\Gamma\{\texttt{def fails } A \texttt{ by builtin}\}\Delta \rightsquigarrow F(\mathit{eval}(A))}$$

$$\frac{A \in \mathrm{dom}(\mathit{eval}) \qquad \Gamma; \Delta \triangleright \texttt{terminates } A}{\Gamma\{\texttt{def terminates } A \texttt{ by builtin}\}\Delta \rightsquigarrow T(\mathit{eval}(A))}$$

**Example:** Under the assumption `succeeds ?x is 2 + 3` the derivation step

```
def succeeds ?x is 2 + 3 by builtin
```

derives the fomula `?x = 5`.

**See also:** The functions $S$, $F$ and $T$ are defined on page 46.

**Inference rule:** atom introduction (built-in) 0.2%

**Syntax:**

`succeeds` *atom*
`fails` *atom*
`terminates` *atom*

**Description:** Assume that $A$ is a built-in atom such that $A \in \mathrm{dom}(\mathit{eval})$ and $\mathit{eval}(A) = G$. If $S(G)$ is derivable from $\Delta$ with length $n$ then `(succeeds` $A$`)` is derivable with length $n+1$. If $F(G)$ is derivable from $\Delta$ with length $n$ then `(fails` $A$`)` is derivable with length $n+1$. If $T(G)$ is derivable from $\Delta$ with length $n$ then `(terminates` $A$`)` is derivable with length $n+1$.

**Formal rules:**

$$\frac{A \in \mathrm{dom}(\mathit{eval}) \qquad \Gamma; \Delta \rhd^n S(\mathit{eval}(A))}{\Gamma; \Delta \rhd^{n+1} \texttt{succeeds}\ A}$$

$$\frac{A \in \mathrm{dom}(\mathit{eval}) \qquad \Gamma; \Delta \rhd^n F(\mathit{eval}(A))}{\Gamma; \Delta \rhd^{n+1} \texttt{fails}\ A}$$

$$\frac{A \in \mathrm{dom}(\mathit{eval}) \qquad \Gamma; \Delta \rhd^n T(\mathit{eval}(A))}{\Gamma; \Delta \rhd^{n+1} \texttt{terminates}\ A}$$

**See also:** The functions $S$, $F$ and $T$ are defined on page 46.

### Axioms for built-in predicates

Unfortunately the above rules for built-in predicates are not enough. There is nothing that corresponds to the induction principle for user-defined predicates. The only way to use statements like

```
all [x,y]: succeeds ?x =< ?y => ?x = ?y \/ succeeds ?x < ?y,
```

is by postulating them as axioms, for example as:

```
:- axiom(leq:less,
all [x,y]: succeeds ?x =< ?y => ?x = ?y \/ succeeds ?x < ?y
).
```

An axiom is treated like a lemma without proof. The formula is added to the internal data base of LPTP as an axiom and it is written to the '`.thm`' file. It can then be referred to in derivation steps. For example:

```
[succeeds ?x =< ?y by gap,
 ?x = ?y \/ succeeds ?x < ?y by axiom(leq:less)]
```

Axioms cannot be arbitrary. They must be true in all structures that are least fixed points of the standard operator associated to a logic program (see [16]). The LPTP cannot check whether this condition is satisfied or not, therefore *the user is responsible for it.* To give an idea what kind of axioms are true in fixed point structures we give some examples. First we define an auxiliary predicate that is needed in some axioms:

```
arithmetic(X) :- integer(X).
arithmetic(X + Y) :- arithmetic(X), arithmetic(Y).
arithmetic(X - Y) :- arithmetic(X), arithmetic(Y).
arithmetic(X * Y) :- arithmetic(X), arithmetic(Y).
arithmetic(X / Y) :- arithmetic(X), arithmetic(Y).
arithmetic(- X) :- arithmetic(X).
```

Here, is a list of possible axioms for built-in preciates:

```
:- axiom((is):integer,
all [x,y]: succeeds ?x is ?y & succeeds arithmetic(?y) =>
 succeeds integer(?x)
).

:- axiom((is):function,
all [x1,x2,y]: succeeds ?x1 is ?y & succeeds ?x2 is ?y =>
 ?x1 = ?x2
).

:- axiom((is):existence,
all y: succeeds arithmetic(?y) => (ex x: succeeds ?x is ?y)
```

```
).

:- axiom((is):termination,
all [x,y]: succeeds arithmetic(?y) => terminates ?x is ?y
).

:- axiom(integer:gr,
all x: succeeds integer(?x) => gr(?x)
).

:- axiom(leq:termination,
all [x,y]: succeeds arithmetic(?x) &
 succeeds arithmetic(?y) => terminates ?x =< ?y,
).

:- axiom(less:termination,
all [x,y]: succeeds arithmetic(?x) &
 succeeds arithmetic(?y) => terminates ?x < ?y,
).

:- axiom(leq:reflexive,
all x: succeeds arithmetic(?x) => succeeds ?x =< ?x
).

:- axiom(less:leq,
all [x,y]: succeeds ?x < ?y => succeeds ?x =< ?y
).

:- axiom(leq:less,
all [x,y]: succeeds ?x =< ?y => ?x = ?y \/ succeeds ?x < ?y
).

:- axiom(less:success:leq:failure,
all [x,y]: succeeds ?x < ?y => fails ?y =< ?x
).

:- axiom(less:failure:leq:success,
all [x,y]: fails ?x < ?y => succeeds ?y =< ?x
).

:- axiom(leq:less:transitive,
all [x,y,z]: succeeds ?x =< ?y & succeeds ?y < ?z =>
 succeeds ?x < ?z
).

:- axiom(less:leq:transitive,
```

```
all [x,y,z]: succeeds ?x < ?y & succeeds ?y =< ?z =>
 succeeds ?x < ?z
).
```

In the presence of built-in predicates the statement (T4) of the beginning of this chapter has to be read as follows:

(T4) If $P \vdash$ `terminates` $A$, then the Prolog computation of $A$ terminates independently of the order of clauses in the Program. Moreover, the computation of $A$ does not flounder. This means, that if a built-in atom $B$ is called then $B$ belongs to the domain of *eval* and can be evaluated. If a negated goal is called, then it is ground.

Of course, this theorem is only true if the axioms for the built-in axioms satisfy the condition mentioned above.

# Chapter 5

# Commands and Tactics

In this chapter we introduce the different modes of LPTP and say how they can be changed. Then we complete the list of commands of LPTP. Finally we explain the tactics of LPTP.

## 5.1 Commands

The commands that are used in proof files have already been described in Chapter 2. The commands for modes and flags have been described in the previous section. The remaining commands of LPTP are listed below. Some of them have already been introduced earlier, but we repeat them here for a complete survey.

`?- def(`*formula*`).`
: The command `def(`$\varphi$`)` prints the definition of $\varphi$ to the standard output. If $\varphi$ is of the form `succeeds` $A$, `fails` $A$ or `terminates` $A$, then the defining formula of $\varphi$ is printed. The atom $A$ can be user-defined or built-in. If $\varphi$ is an atom of the form $R(\vec{t})$, where $R$ is an abbreviation, then the definition of $R$ is expanded. See also the Emacs command '`C-c i d`'.

`?- facts(`*reference*`).`
: This command prints all facts that are stored in the internal database of LPTP and match *reference* to the standard output. The argument *reference* of the command can be a name or a colon separated list of names. The command prints all the facts of the internal database that contain all the components of *reference* in their names. For example, the command `facts(list)` prints all facts about `list`; the command `facts(length:nat)` prints all facts about `length` and `nat`. See also the Emacs command '`C-c i l`'.

`?- depends(`*fact*`(`*reference*`)).`
: This command prints all facts on which *fact*(*reference*) depends to the

standard output. For this command the flag `write_dependencies` has to be set. See p. 121 for more details.

`?- mark(`*formula*`).`
: Several tactics use a marked formula. The command `mark(`$\varphi$`)` markes the formula $\varphi$ such that it can be used by tactics later. See Sect. 5.2 for more details. See also the Emacs command '`C-c i m`'.

`?-` *formula* `by [`*tactic*`,`*options*`,...].`
: This command tries to prove *formula* by *tactic*. See Sect. 5.2 for more details on tactics.

## Examples

In the following example we assume that the clauses and the theorems for lists have been loaded with the following commands:

```
?- needs_gr($(lib)/list/list).
?- needs_thm($(lib)/list/list).
```

The first example shows the expansion of an abbreviation.

```
?- def(sub(?l1,?l2)).

all x: succeeds member(?x,?l1) => succeeds member(?x,?l2)
```

The next example computes the defining form of a user-defined predicate.

```
?- def(succeeds append([?x|?l0],?l1,?l2)).

ex l3: ?l2 = [?x|?l3] & succeeds append(?l0,?l1,?l3)
```

The next two examples compute the definitions of built-in predicates.

```
?- def(fails call(list,?l)).

fails list(?l)

?- def(succeeds ?x is 2 + 5).

?x = 7
```

The next example lists all facts about `member`.

```
?- facts(member).

Lemma (member:termination)
all [x,l]: succeeds list(?l) => terminates member(?x,?l).

...
```

The next example shows how to use tactics as interactive commands.

```
?- p & (q \/ r) => (p & q) \/ (p & r) by [auto(5),r(20)].

======
assume(p & (q \/ r),
 cases(q,
  [],
  r,
  [],
  p & q \/ p & r),
 p & q \/ p & r)
======
```

The command '`by`' is only used for small examples. In real proofs a pseudo derivation step is applied. The next section explains the details.

## 5.2 Tactics

The most convenient way to use tactics is together with the LPTP Emacs mode which is described in Appendix A. If this mode is installed correctly, then proof files with the extenstion '`.pr`' let Emacs go into LPTP mode. In this mode there is a menu on on the menubar which lists the tactics.

Tactics are implemented as pseudo derivation steps similar to the `gap` and `because` derivation step. If LPTP reaches a tactic derivation step, it tries to apply the tactics using the current list of assumptions $\Gamma$, the marked formula (if it exists), and the current list of already derived formulas $\Delta$. As a side effect LPTP writes a derivation to standard output. The user can then replace the pseudo derivation step by the derivation computed by LPTP. In Emacs mode this is done with the *get* command. Some tactics use the so-called marked formula. Sometimes, there are more than one possible ways to apply a tactics. In this case, the option `more` forces LPTP to backtrack for alternative possibilities.

We start with the general description of a tactic derivation step and the possible options. The single tactics will be explained afterwards alphabetically. The first tactic `auto` is conceptually different from the remaining tactics. It is used for automatic derivations. The other tactics are used for interactive derivation. They only compute the next two or three obvious steps in a derivation.

**Derivation step:** tactic

**Syntax:**

*formula* `by []`
*formula* `by [`*tactic*`]`
*formula* `by [`*tactic*`,`*option*`,...]`

**Description:** The derivation step

$\varphi$ `by [`*tactic*`,`*option*`,...]`

derives the formula $\varphi$. As a side effect, LPTP tries to prove $\varphi$ using *tactic*. The (partial) derivation is written to standard output.

**Options:**

`more`
: This option tells LPTP to ask the user whether he wants alternative solutions.

`l(`$n$`)`
: This option tells LPTP to indent the output by $n$ columns. Default is `l(0)`. In Emacs mode, the insertion of the appropriate `l(`$n$`)` option is done automatically. The `l` stands for *left margin*.

`r(`$n$`)`
: This option tells LPTP that the width of the screen is $n$ columns. The number $n$ is used for breaking formulas at appropriate places. Default is `r(76)`. The `r` stands for *right margin*.

**Formal rule:**

$$\overline{\Gamma\{\varphi \texttt{ by [...]}\}\Delta \rightsquigarrow \varphi}$$

**Example:** The derivation step

```
sub(?l1,?l2) by [unfold,l(4)]
```

has — as a side effect — the following output:

```
======
   [assume(succeeds member(?x,?l1),
     succeeds member(?x,?l2) by gap,
     succeeds member(?x,?l2)),
    sub(?l1,?l2) by introduction(sub,2)]
======
```

| **Tactic:** automatic | `auto(`$n$`)` |
|---|---|

**Description:** The tactics

$\varphi$ `by [auto(`$n$`),...]`
$\varphi$ `by [auto(`$n$`),more,...]`

causes LPTP to search for an automatic proof of $\varphi$ from the current list of assumptions $\Gamma$ and the list of already derived formulas $\Delta$. The integer $n$ is a bound for the depth of the search. The option `more` causes LPTP to search for alternative derivations. In Emacs one control the search depth using the argument `C-u` $n$.

**Emacs commands:** `C-c i a`, `C-u` $n$ `C-c i a`.

| **Tactic:** case splitting | `case` |
|---|---|

**Description:** The tactics

$\varphi$ `by [case,...]`
$\varphi$ `by [case,more,...]`

pick a disjunction $\psi$ from the list of already derived formulas $\Delta$ and tries to prove $\varphi$ by a case splitting on that disjunction. If the disjunction $\psi$ is of the form $\psi_1$ `\/` $\psi_2$ it inserts at least the following derivation step:

`case(`$\psi_1$`,`
$\varphi$ `by gap,`
$\psi_2$`,`
$\varphi$ `by gap,`
$\varphi$`)`

The option `more` causes LPTP to pick a different disjunction. If a disjunction has been marked using '`C-c i m`', then this disjunction is used.

**Emacs commands:** `C-c i c`, `C-u C-c i c`

**Tactic:** completion (fixed point) `comp`

**Description:** The tactics

$\varphi$ `by [comp,...]`
$\varphi$ `by [comp,more,...]`

pick a formula $\psi$ of the form `(succeeds` $A$`)` or `(fails` $A$`)` or `(terminates` $A$`)` from the list of already derived formulas $\Delta$; then they insert the derivation step

`def` $\psi$ `by completion`

and try to derive $\varphi$ from $D^P(\psi)$. The option `more` causes LPTP to pick a different $\psi$ from $\Delta$. If a formula has been marked using '`C-c i m`', then this formula is used.

**Emacs commands:** `C-c i o`, `C-u C-c i o`.

**Tactic:** expansion of an abbreviation `elim`

**Description:** The tactics

$\varphi$ `by [elim,...]`
$\varphi$ `by [elim,more,...]`

pick an atom $R(\vec{t})$ from the list of already derived formulas $\Delta$ such that $R$ is a definded predicate symbol. Assume that $R$ has the following definition:

`:- definition_pred(`$R$`,`$n$`,all [`$\vec{x}$`]:` $R(\vec{x})$ `<=>` $\psi(\vec{x})$`).`

Then the tactics expand $R(\vec{t})$ into the formula $\psi(\vec{t})$. They insert the derivation step

$\psi(\vec{t})$ `by elimination(`$R$`,`$n$`)`

and try to prove $\varphi$ from $\Delta$ plus $\psi(\vec{t})$. The option `more` causes LPTP to pick another atom $R(\vec{t})$ from $\Delta$. If an atom $R(\vec{t})$ has been marked using '`C-c i m`', then this atom is used.

**Emacs commands:** `C-c i e`, `C-u C-c i e`.

| **Tactic:** existence elimination | `ex` |
|---|---|

**Description:** The tactics

`` $\varphi$ by [ex,...] ``
`` $\varphi$ by [ex,more,...] ``

pick an existentially quantified formula `(ex [`$\vec{x}$`]:` $\psi$`)` from the list of already derived formulas $\Delta$ and try to prove $\varphi$ by eliminating the existential quantifier and assuming that there are $\vec{x}$ such that $\psi$ holds. They insert at least the following derivation steps:

`exist(`$\vec{x}$`,`
$\psi$`,`
`` $\varphi$ by gap, ``
$\varphi$`)`

The option `more` causes LPTP to pick a different existentially quantified formula from $\Delta$. If an existentially quantified formula has been marked using '`C-c i m`', then this formula is used.

**Emacs commands:** `C-c i x`, `C-u C-c i x`.

| **Tactic:** fact | `fact` |
|---|---|

**Description:** The tactic

$\varphi$ `by [fact,...]`

tries to derive $\varphi$ by a lemma, theorem, corollary or axiom that is stored in the internal database. As a result it prints a derivation step like the following:

$\varphi$ `by lemma(append:termination)`

Note, that the user does not have to remeber the names of the theorems. LPTP will find them automatically. Sometimes, it even changes $\varphi$. For example, if $\varphi$ is the equation $s$ `=` $t$, then the result can be the following:

$t$ `=` $s$ `by lemma(plus:successor)`

Thus, LPTP has commuted $s$ and $t$.

In a first run, LPTP tries to match $\varphi$ against the head of one of the stored facts. If this does not yield anything, then LPTP goes through the whole database and tries each fact in turn for deriving $\varphi$. The second run can take some time.

**Example:** Assume that the `corollary(lh:cons)` refers to the formula

```
all [x,l]:succeeds list(?l) => lh([?x|?l]) = s(lh(?l))
```

Then we can write:

```
assume(succeeds list(?l),
 lh([?x|?l]) = s(lh(?l)) by [fact],
 ...)
```

The system will answer with:

```
lh([?x|?l]) = s(lh(?l)) by corollary(lh:cons)
```

**Emacs command:** `C-c i f`.

| **Tactic:** induction | `ind` |
|---|---|

**Description:** The tactic

$\varphi$ `by [ind,...]`

generates the induction scheme for $\varphi$. In order that the induction scheme can be created, the formula $\varphi$ must have the following form:

`all [`$\vec{x}$`]: succeeds` $R(\vec{x})$ `=>` $\psi$

Or, it must be a conjunction of such formulas:

`(all [`$\vec{x}$`]: succeeds` $R(\vec{x})$ `=>` $\psi$`) & ...`

The predicate $R$ must be user-defined, and the clauses for $R$ must be loaded into the internal database.

Sometimes the formula that has to be proved by induction is not in the right form. For example, let $\varphi$ be the formula

`all [`$\vec{x}, \vec{y}$`]: succeeds` $R(\vec{x})$ `&` $\psi$ `=>` $\chi$`.`

Then the tactic `ind` does the following. It creates the induction scheme for the following formula $\varphi'$ which is equivalent to $\varphi$:

`all [`$\vec{x}$`]: succeeds` $R(\vec{x})$ `=> (all [`$\vec{y}$`]` $\psi$ `=>` $\chi$`).`

**Example:** The tactic

```
all [l1,l2,l3]: succeeds list(?l3) =>
  terminates append(?l1,?l2,?l3) by [ind]
```

generates the following derivation:

```
induction(
 [all l3: succeeds list(?l3) =>
    (all [l1,l2]: terminates append(?l1,?l2,?l3))],
 [step([],
   [],
   [],
   all [l1,l2]: terminates append(?l1,?l2,[])),
  step([x,l],
   [all [l1,l2]: terminates append(?l1,?l2,?l),
    succeeds list(?l)],
   [],
   all [l1,l2]: terminates append(?l1,?l2,[?x|?l]))])
```

It generates the induction scheme for the following formula:

```
all l3: succeeds list(?l3) =>
  (all [l1,l2]: terminates append(?l1,?l2,?l3))
```

Then it derives automacally the original formula:

```
all [l1,l2,l3]: succeeds list(?l3) =>
  terminates append(?l1,?l2,?l3)
```

**Emacs command:** `C-c i i`.

| **Tactic:** induction (quantifier-free) | `indqf` |
|---|---|

**Description:** If $\varphi$ is the formula

`all [`$\vec{x},\vec{y}$`]: succeeds` $R(\vec{x})$ `&` $\psi$ `=>` $\chi$.

then the tactic

$\varphi$ `by [indqf,...]`

generates the induction scheme for the following formula:

`all [`$\vec{x}$`]: succeeds` $R(\vec{x})$ `=> (`$\psi$ `=>` $\chi$`)`.

**Example:** The tactic

```
all [y,l]: succeeds list(?l) =>
  terminates member(?y,?l) by [indqf]
```

generates the following derivation:

```
induction(
 [all l: succeeds list(?l) => terminates member(?y,?l)],
 [step([],[],[],terminates member(?y,[])),
  step([x,l],
   [terminates member(?y,?l),
    succeeds list(?l)],
   [],
   terminates member(?y,[?x|?l]))])
```

**Emacs command:** `C-c i q`.

**Tactic:** totality `tot`

**Description:** The tactics

$\varphi$ `by [tot,...]`
$\varphi$ `by [tot,more,...]`

pick a formula of the form `(terminates` $A$`)` from the list of already derived formulas $\Delta$. Then they make a case splitting with `(succeeds` $A$`)` and `(fails` $A$`)`. At least they print the following derivation step:

`cases(succeeds` $A$`,`
$\varphi$ `by gap,`
`fails` $A$`,`
$\varphi$ `by gap,`
$\varphi$`)`

The option `more` causes LPTP to pick a different termination atom from $\Delta$ and to make a different case splitting on success and failure. If a termination atom has been marked using '`C-c i m`', then this atom is used.

**Emacs commands:** `C-c i t`, `C-u C-c i t`.

**Tactic:** unfold (success) `unfold`

**Description:** The tactic

`succeeds` $A$ `by [unfold,...]`

computes the defining formula $D^P$(`succeeds` $A$) and tries to derive this formula. At least it prints the following derivation:

[$D^P$(`succeeds` $A$) `by gap,`
`succeeds` $A$ `by completion]`

**Emacs command:** `C-c i u`.

**Tactic:** unfold (failure) `unfold`

**Description:** The tactic

`fails` $A$ `by [unfold,...]`

computes the defining formula $D^P$(`fails` $A$) and tries to derive this formula. At least it prints the following derivation:

[$D^P$(`fails` $A$) `by gap,`
`fails` $A$ `by completion]`

**Emacs command:** `C-c i u`.

**Tactic:** unfold (termination) `unfold`

**Description:** The tactic

`terminates` $A$ `by [unfold,...]`

computes the defining formula $D^P$(`terminates` $A$) and tries to derive this formula. At least it prints the following derivation:

[$D^P$(`terminates` $A$) `by gap,`
`terminates` $A$ `by completion]`

**Emacs command:** `C-c i u`.

**Tactic:** unfold (abbreviation) `unfold`

**Description:** Assume that $R$ has the following definition:

`:- definition_pred(`$R$`,`$n$`,all [`$\vec{x}$`]:` $R$`(`$\vec{x}$`) <=>` $\varphi$`(`$\vec{x}$`)).`

Then the tactic

$R$`(`$\vec{t}$`) by [unfold,...]`

expands the atom $R(\vec{t})$ into the formula $\varphi(\vec{t})$. It then tries to prove $\varphi(\vec{t})$. At least it prints the following derivation:

[$\varphi(\vec{t})$ `by gap,`
$R(\vec{t})$ `by introduction(`$R$`,`$n$`)`

**Emacs command:** `C-c i u`.

| **Tactic:** unfold (termination of a conjunction) | `unfold` |
|---|---|

**Description:** The tactic

`terminates (`$G_1$ `&` $G_2$ `& ... &` $G_n$`) by [unfold,...]`

does the following. It tries to prove $T(G_1)$ and the formula $T($`(`$G_2$ `& ... &` $G_n$`)` (see page 46). At least it prints the following derivation:

```
[T(G1) by gap,
 T(G2 & ... & Gn) by gap,
 terminates (G1 & G2 & ... & Gn)]
```

The tactic

`terminates (`$G_1$ `&` $G_2$ `& ... &` $G_n$`) by [unfold,more,...]`

tries to prove the following formula:

`Terminates` $G_1$ `& (succeeds` $G_1$ `=> terminates (`$G_2$ `& ... &` $G_n$`)).`

At least it prints the following derivation:

```
[T(G1) by gap,
 assume(S(G1),
  T(G2 & ... & Gn) by gap,
  T(G2 & ... & Gn)),
 terminates (G1 & G2 & ... & Gn)]
```

**Emacs commands:** `C-c i u`, `C-u C-c i u`.

**Tactic:** unfold (uniqueness) `unfold`

**Description:** Assume that $f$ has the following definition:

```
:- definition_fun(f,n,
 all [x⃗,y]: δ(x⃗) => (f(x⃗) = y <=> γ(x⃗,y)),
 existence by ...,
 uniqueness by ...
).
```

Then the tactic

```
f(t⃗) = s by [unfold,...]
```

does the following. It tries to prove the formulas $\delta(\vec{t})$ and $\gamma(\vec{t}, s)$. At least it prints out the following derivation:

```
[δ(t⃗) by gap,
 γ(t⃗,s) by gap,
 f(t⃗) = s by uniqueness(f,n)]
```

**Emacs commands:** `C-c i u`.

**Tactic:** unfold (existence) `unfold`

**Description:** Assume that $f$ has the following definition:

```
:- definition_fun(f,n,
 all [x⃗,y]: δ(x⃗) => (f(x⃗) = y <=> γ(x⃗,y)),
 existence by ...,
 uniqueness by ...
).
```

Then the tactic

```
γ(t⃗,f(t⃗)) by [unfold,...]
```

does the following. It tries to prove the formula $\delta(\vec{t})$. At least it prints out the following derivation:

```
[δ(t⃗) by gap,
 γ(t⃗,f(t⃗)) by existence(f,n)]
```

**Emacs commands:** `C-c i u`.

| **Tactic:** debug | `debug` |
|---|---|

**Description:** The tactic

*formula* `by [debug,...]`

prints as a side effect the set of protected variables and the list of available formulas to standard output.

**Emacs command:** `C-c i b`.

## 5.3 Flags and Modes

The behavior of the LPTP system is controlled by flags. Flags can be set and unset in the following way:

```
?- set(Flag).
?- unset(Flag).
```

In most cases the user does not have to set the flags individualy but can use the four predefined *modes* of LPTP.

```
?- pedantic.
?- plain.
?- draft.
?- show.
```

LPTP is by default in *pedantic* mode when it is started. It checks everything and reports all irregularities as warnings or error messages. The TEX output files are written and the theorem files as well. The *plain* mode is a little bit faster, since it checks only what is really necessary and does not write the TEX output files. Both, *pedantic* mode and *plain* mode are used to check whole proof files. The *draft* mode is used for checking single proofs. It does not write any output and does not assert the facts that is proved to the database, because in most cases the proof of such a fact contains gap. The *draft* mode is the fastest mode. It ignores errors. The *show* mode prints the names of the theorems on the screen. This is useful because it shows the progress of the system when it is checking a large file.

The four modes *pedantic*, *plain*, *draft* and *show* set and unset certain flags according to the following table:

| | pedantic | plain | draft | show |
|---|---|---|---|---|
| check_everything | yes | no | yes | no |
| fail_on_error | yes | yes | no | no |
| unique_names | yes | no | no | yes |
| assert_facts | yes | yes | no | yes |
| report_because | yes | no | no | no |
| tex_output | yes | no | no | no |
| thm_output | yes | yes | no | no |
| print_names | no | no | no | yes |

The single flags have the following effects when set:

`check_everything`
: The following additional tests are made: if a fact is asserted when reading

a '`.thm`' file, then the fact is checked for syntactical correctness; when formulas are compiled into the internal representation it is checked whether the internal representation is a correct formula. The same is done for derivations.

`fail_on_error`
: In case of an error the system fails after it has printed the error message.

`unique_names`
: If a fact is asserted then it is tested whether the name of the fact is new. Unique names are not really necessary. A proof can be correct even if it contains two different lemmas with the same name.

`assert_facts`
: After a proof has been checked the formula it proves is asserted to the internal database. If the flag `assert_facts` is set, then this is done, even if the proof contains gaps or incorrect derivation steps.

`report_because`
: Prints a warning for *because* gaps in proofs. Sometimes the warnings abouts gaps in proofs are annoying and the user wants to turn them of. This can be done by unsetting the flag `report_because`.

`tex_output`
: If the flag `tex_output` is unset, then no TEX output is written even if the file contains a `tex_file` command.

`thm_output`
: If the flag `thm_output` is unset, then no theorem file is written even if the file contains a `thm_file` command.

`print_names`
: Prints the names of the theorems, lemmas, corollaries and axioms on the screen. This is useful if the user wants to know where the system is.

`write_dependencies`
: If the flag `write_dependencies` is set then the dependencies between theorems, lemmas, corollaries, axioms and function definitions are written to the '`.thm`' file. If the '`.thm`' file is later read into the system using `needs_thm`, then the dependencies are added to the internal database and are printed to the standard output with the command `depends`.

`debug`
: Go into debugging mode. The debugging mode is used to debug the system and not to debug proofs. Use $\varphi$ `by [debug]` for debugging the formula $\varphi$ in a proof, i.e. to see why $\varphi$ is not derivable.

`global_error, set_global_error`
: Internal flags that are set and unset by LPTP and not by the user.

# Chapter 6

# The library

(UNDER CONSTRUCTION)

## 6.1 Natural numbers and arithmetic

Some rules for natural numbers are built-in to LPTP.

(. . . )

See: '`lptp/lib/nat/`'

| **By:** addition | 0.0% |
|---|---|

**Syntax:**

*term* `=` *term* `by addition`

## 6.2 Lists

Some rules for list processing are built-in to LPTP.

(. . . )

See: '`lptp/lib/list/`'

| **By:** concatenation | 1.5% |
|---|---|

**Syntax:**

*term* `=` *term* `by concatenation`

# Chapter 7

# Examples

(UNDER CONSTRUCTION)

## 7.1 Sorting algorithms with call/3

See: '`lptp/lib/builtin/`'

## 7.2 A tautology checker

See: '`lptp/examples/taut/`'

## 7.3 A verified parser for ISO standard Prolog

See: '`lptp/examples/parser/`'

## 7.4 Algorithms for AVL-trees

See: '`lptp/examples/avl/`'

## 7.5 Min-max and alpha-beta pruning

See: '`lptp/examples/alpha/`'

## 7.6 A union-find based unification algorithm

See: '`lptp/examples/mgu/`'

# Appendix A

# Emacs mode

LPTP mode customizes Emacs for editing formal proofs. It changes the `TAB`, `DEL` and `LFD` keys and defines '`C-c i` $x$' commands for the interaction with an inferior Prolog process that runs LPTP. The output created by LPTP is automatically copied from the Prolog buffer to the edit buffer. LPTP mode provides a menubar menu and it makes it possible to select a formula by double-clicking on it with the left mouse button.

## Installing LPTP mode

The following lines in the file '`.emacs`' will make Emacs use LPTP mode automatically when editing files with a '`.pr`' extension:

```
(autoload 'lptp-mode "~/lptp/etc/lptp-mode"
        "Major mode for editing formal proofs" t)

(setq auto-mode-alist
        (cons '("\\.pr$" . lptp-mode) auto-mode-alist))
```

The string `"~/lptp/etc/lptp-mode"` in the first line is the path to the file '`lptp-mode.el`'. It has to be changed accordingly. If the directory of the file '`lptp-mode.el`' is already in the Emacs load path, then one can use the string `"lptp-mode"`.

The Emacs lisp file '`lptp-mode.el`' can be compiled with the command '`M-x byte-compile-file`'. Emacs will then use the byte-code file with the '`.elc`' extension.

The following two variables determine the Prolog interpreter and the command that is used to load the LPTP system into the Prolog interpreter.

`prolog-program-name`
: A variable that contains the command for starting Prolog. Example values are: `"sicstus"`, `"quintus"`, `"cprolog"`.

`lptp-start-string`
: A variable that contains the command for loading the LPTP system into Prolog interpreter. Example values are:
`"load(lptp)"`, `"[lptp]"`, `"consult('lptp.pl')"`.

If the LPTP system has been compiled so that it extends a Prolog interpreter (eg. GNU Prolog), then the variables have to be set in the following way:

```
(defvar prolog-program-name "/home/staerk/lptp/bin/lptp"
    "Program name for invoking an inferior Prolog process.")

(defvar lptp-start-string ""
    "The Prolog command to load the LPTP system.")
```

These variables can be changed in the file '`lptp-mode.el`' or they can be set using a so-called hook in the '`.emacs`' file:

```
(add-hook 'lptp-mode-hook
  (lambda ()
    (setq prolog-program-name "/usr/local/bin/sicstus")
    (setq lptp-start-string
          "consult('/home/staerk/lptp/src/lptp.pl')")))
```

You can also use a so called 'local variables list' at the end of a file that you want to edit in LPTP mode. Include the following lines as comments at the end of the file:

```
% Local Variables:
% mode: lptp
% End:
```

These lines tell Emacs to use LPTP mode.

## Editing formal proofs

`C-c C-a` (`lptp-beginning-of-formula`)
: Go to the beginning of this formula or to the beginning of this proof step.

`C-c C-e` (`lptp-end-of-formula`)
: Go to the end of this formula or to the end of this proof step.

`C-c C-n` (`lptp-next-formula`)
: Go to the beginning of the next formula or the beginning of the next proof.

`C-c C-p` (`lptp-previous-formula`)
: Go to the beginning of the previous formula or the beginning of the previous proof step.

`C-c C-m` (`lptp-mark-fact`)
: Put point at the beginning and mark at the end of this fact. A fact is an axiom, corollary, lemma or theorem.

`LFD` (`newline-and-indent`)
: Insert a newline, then indent.

`TAB` (`indent-for-tab-command`)
: Indent this line in the proper way.

`double-mouse-1` or `C-c C-@` (`lptp-mark-formula`)
: Put point at the beginning and mark at the end of this formula or this proof step and copy it into the kill ring. The formula marked is the one that contains point or follows point.

`C-c i [` (`lptp-insert-brackets`)
: Insert a pair of brackets around this formula and indent it in the proper way.

`C-c i r` (`ltpt-replace-ff-by-gap`)
: Replace '`ff by gap`' by the previous formula.

`C-c i ~` (`lptp-backup-buffer`)
: Save buffer and make the previous version into a backup file.

The commands '`C-c C-n`' and '`C-c C-p`' are used to step quickly through files that contain formal proofs. For example, put point to the first '`:-`' and type '`C-c C-n`'. The commands are also useful to jump from one step to another in an induction proof. For example, put point to the word `step` and type '`C-c C-n`'.

If you wish to operate on the current lemma, theorem or corollary, use '`C-c C-m`' which puts point at the beginning and mark at the end of the current fact. For example, this is the easiest way to get ready to move the fact to a different place in the text.

To indent a line, use the `TAB` command. No matter where in the line you are when you type `TAB`, it aligns the line as a whole in an appropriate way. The `TAB` command is useful to detect syntax errors early. In a list of formulas the `TAB` command scans all the way back to the first formula of the list to calculate the right indentation. This is not very efficient but has shown to be useful in practise.

The `double-mouse-1` command sets the region around the item you click on. The text is put into the kill ring so that you can yank it with `mouse-2`. If you double-click on an `all` or `ex` quantifier you select the whole scope of that quantifier. If you double click on an opening bracket '`[`' you select the whole list of items which starts at the '`[`'. If you double-click on a key word like `assume`, `case`, `contra`, `induction` or `step` you select the corresponding proof step. Turn on region highlighting with the command '`M-x transient-mark-mode`' to see the effect of double-clicking. You can also put the command

```
(transient-mark-mode 1)
```

in your '`.emacs`' file.

The command '`C-c i [`' is designed to facilitate a style of editing which keeps brackets balanced at all times. '`C-c i [`' inserts a pair of square brackets around the current formula. It leaves point after the open bracket. It is used in the following situation:

```
assume(φ,
 ψ by gap,
 χ)
```

Suppose that point is at the beginning of $\psi$. Then '`C-c i [`' changes the proof fragment to the following:

```
assume(φ,
 [
  ψ by gap],
 χ)
```

Now one can insert a new formula in the line above $\psi$. This works also if the formula $\psi$ extends over several lines.

At the beginning proofs are usually empty. In a first step the empty proof (`ff by gap`) has to be replaced by the formula of the lemma we want to prove. The command '`C-c i r`' is used in the following situation:

```
:- lemma(reference,
 φ,
 ff by gap
).
```

Suppose that point is at the beginning of '`ff by gap`'. Then '`C-c i r`' changes the text fragment to the following:

```
:- lemma(reference,
 φ,
 φ by []
).
```

Now one can send the buffer to a running LPTP process or one can use tactics command to further edit the proof.

Usually, Emacs makes a backup for a file only the first time the file is saved. The command '`C-c i ~`' causes Emacs to make so-called numbered backups with the extensions `.~1~`, `.~2~`, and so on. To prevent unlimited consumption of disk space, Emacs deletes numbered backup versions automatically. The two variables `kept-old-versions` and `kept-new-versions` control this deletion. In longer proofs, we recommend to make backup files frequently.

## Running LPTP

`M-x run-lptp`
: Run a new LPTP process in the buffer `*lptp*`.

`M-x run-lptp-other-frame`
: Run a new LPTPprocess in the buffer `*lptp*` in a new frame.

`M-x lptp-quit`
: Quit the LPTP process and kill the buffer `*lptp*`.

`C-c i s` (`lptp-save-send-buffer`)
: Save buffer and send it to the LPTP process.

`C-c i g` (`lptp-get`)
: Replace the current formula by the output from the LPTPprocess. The point should be at the topmost level in the formula.

`C-c i d` (`lptp-definition`)
: Take the atom that is in the region and display its definition in the `*lptp*` buffer.

`C-c i l` (`lptp-list-facts`)
: Lists all the facts about the symbol that is in the region in the `*lptp*` buffer.

`C-c i m` (`lptp-mark-assumption`)
: Take the formula that is in the region and send it to the `*lptp*` buffer as marked assumption.

The commands '`C-c i s`' and '`C-c i g`' communicate with a Prolog process that has been started with '`M-x run-lptp`' or '`M-x run-lptp-other-frame`'. '`C-c i s`' saves the buffer, if it has been modified and sends it to the Prolog process. More precisely, it sends the command `consult('`*buffer-name*`')` to the Prolog process. The '`C-c i s`' command is not used very often, since it is included in the tactics command.

The '`C-c i g`' commands replaces the current formula with the output created by the system. This command is smart and removes the enclosing brackets of the LPTP output if this is necessary. For example, suppose that in the edit buffer we have

```
assume(φ0,
 [φ1,
  φ2],
 φ3)
```

and that point is at the beginning of formula $\varphi_1$. Suppose that the output in the `*lptp*` buffer looks as follows:

```
======
  [ψ0,
   ψ1]
======
```

Then the '`C-c i g`' command changes the edit buffer to

```
assume(φ0,
 [ψ0,
  ψ1,
  φ2],
 φ3)
```

Note, that the formulas $\psi_0$ and $\psi_1$ are even automatically moved one column to the left.

The command '`C-c i d`' displays the definition of an atom in the `*lptp*` buffer if there is a definition for the atom. It sends the command `def(`*region*`)` to the Prolog process.

The command '`C-c i l`' lists all facts about a symbol in the `*lptp*` buffer. It sends the command `facts(`*region*`)` to the Prolog process. A fact is a theorem, lemma, corollary or an axiom.

The command '`C-c i m`' markes a formula for later use in a tactics command. It sends the command `mark(`*region*`)` to the Prolog process.

## Tactics

`C-c i a` (`lptp-tactic-auto`)
: Try to prove this formula automatically. With a numeric argument change the default search depth to that argument.

`C-c i c` (`lptp-tactic-case`)
: Try to prove this formula by case splitting. With an argument, pick other possible disjunctions.

`C-c i o` (`lptp-tactic-comp`)
: Try to prove this formula by expanding a formula of the form `succeeds` *atom*, `fails` *atom* or `terminates` *atom*. If a formula has been marked using `C-c i m`, then use the marked formula. With an argument, pick other possible formulas.

`C-c i e` (`lptp-tactic-elim`)
: Try to prove this formula by expanding a defined predicate (eliminate the predicate). If a formula has been marked using '`C-c i m`', then use the marked formula. With an argument, pick other possible formulas.

`C-c i x` (`lptp-tactic-ex`)
: Try to prove this formula by elimination of an existential formula. If a formula has been marked using '`C-c i m`', then use the marked formula. With an argument, pick other possible existentially quantified formulas.

`C-c i f` (`lptp-tactic-fact`)
: Try to prove this formula by a fact. A fact is an axiom, a lemma, a theorem or a corollary.

`C-c i i` (`lptp-tactic-ind`)
: Try to prove this formula by induction.

`C-c i q` (`lptp-tactic-indqf`)
: Try to prove this formula by quanitfier-free induction.

`C-c i t` (`lptp-tactic-tot`)
: Try to prove this formula by case splitting on a termination atom. If a formula has been marked using '`C-c i m`', then use the marked formula. With an argument, pick other possible termination atoms.

`C-c i u` (`lptp-tactic-unfold`)
: Try to prove this formula by unfolding it. With an argument, try different possibilities.

`C-c i b` (`lptp-tactic-debug`)
: Print the protected variables and the list of available formulas at this point in the proof.

The tactics commands work all in a similar way. They insert at the end of the current formula the string '`by [`*name*`,l(`$n$`)]`', where *name* is taken from `lptp-tactic-`*name* and $n$ is the indentation of the formula. Then they save the buffer and send it to the Prolog process. If there exists already a '`by` *string*' extension, it is replaced.

With an argument, the commands insert '`by [`*name*`,more,l(`$n$`)]`'. For example, if '`C-c i c`' picks the wrong disjunction for case splitting, you can use '`C-u C-c i c`' and force LPTP to backtrack and to display other possibilities.

The '`C-c i a`' works different. It inserts '`by [auto(5),more,l(`$n$`)]`' at the end of the formula. If you want to increase the search depth from 5 to 9, use '`M-9 C-c i a`' or '`C-u 9 C-c i a`'.

## The LPTP menu

The LPTP menu is placed in the menu bar. The items of the menu are printed in table A.1. You can customize the menu in the '`lptp-mode.el`' file.

## Memorizing Emacs commands

The '`C-c i` $x$' commands can easily be memorized:

```
Send buffer      (C-c i s)
Get output       (C-c i g)
--------------------------
Tactic: auto     (C-c i a)
Tactic: case     (C-c i c)
Tactic: comp     (C-c i o)
Tactic: elim     (C-c i e)
Tactic: ex       (C-c i x)
Tactic: fact     (C-c i f)
Tactic: ind      (C-c i i)
Tactic: indqf    (C-c i q)
Tactic: tot      (C-c i t)
Tactic: unfold   (C-c i u)
--------------------------
Print definition (C-c i d)
List facts       (C-c i l)
Debug            (C-c i b)
--------------------------
Mark formula     (C-c i m)
Insert brackets  (C-c i [)
Backup buffer    (C-c i ~)
--------------------------
Run LPTP
Quit LPTP
```

Table A.1: The Emacs LPTP menu.

**Commands for tactics:**

`a` — automatic
`b` — debug
`c` — case splitting
`e` — elimination of defined predicate
`f` — fact
`i` — induction
`o` — completion
`q` — quantifier-free induction
`t` — totality
`u` — unfold
`x` — existential elimination

**Commands for interaction with LPTP:**

`d` — definition
`g` — get output
`l` — list all facts
`m` — mark assumption
`s` — save and send buffer

**Editing commands:**

`r` — replace by previous
`[` — insert brackets
`~` — backup

## Creating tags tables for proof files

In large multi-file proofs it is convenient to jump directly to the axioms, lemmas, theorems and corollaries using Emacs tags tables and the '`M-.`' command (`find-tag`). To create a tags table for '`.pr`' files use the `lptp-tags` shell script in the `lptp/etc` directory. Note that this script needs Emacs version 19.30 or later. Go to the directory that contains the '`.pr`' files and type the command

```
lptp-tags *.pr
```

This command will create a file `TAGS` listing the names of the component files and the names and positions of all the axioms, lemmas, corollaries and theorems. You can use the `M-TAB` command (`complete-tag`) to perform tags completion on the text around point and '`M-.`' (`find-tag`) to find the tag's definition. The first command is used to complete the name of a lemma, theorem or corollary and the second command is used to find the lemma, theorem or corollary. See the subsection on tags in the Emacs manual for more information. The `lptp-tags` script is an abbreviation for

```
etags
  --language=none
  --regex='/:- \(axiom\|lemma\|corollary\|theorem\)[^,]*/'
```

Note that earlier versions of the `etags` program do not have the possibility to define regular expressions.

*Please do not define C-c ⟨letter⟩ as a key in your major modes. These sequences are reserved for users; they are the only sequences reserved for users, so we cannot do without them. Instead, define sequences consisting of C-c followed by a non-letter. These sequences are reserved for major modes. Changing all the major modes in Emacs 18 so they would follow this convention was a lot of work. Abandoning this convention would waste that work and inconvenience the users.*

# Bibliography


[1] K. R. Apt and D. Pedreschi. Proving termination of general Prolog programs. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 265–289. Springer-Verlag, Lecture Notes in Computer Science 526, 1991. 88

[2] K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993. 88

[3] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978. 66

[4] J.-Y. Girard. *Proof Theory and Logical Complexity*. Bibliopolis, Napoli, 1987. 7

[5] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994. 48

[6] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland, Amsterdam, 1974. 7

[7] W. Pohlers. *Proof theory: an introduction*, volume 1407 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1989. 7

[8] K. Schütte. *Proof Theory*. Springer-Verlag, Berlin, 1977. 7

[9] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. of Logic Programming*, 29(1–3):17–64, 1996. 87

[10] R. F. Stärk. The declarative semantics of the Prolog selection rule. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science, LICS '94*, pages 252–261, Paris, France, July 1994. IEEE Computer Society Press. 7

[11] R. F. Stärk. First-order theories for pure Prolog programs with negation. *Archive for Mathematical Logic*, 34(2):113–144, 1995. 7, 86

[12] R. F. Stärk. A transformation of propositional Prolog programs into classical logic. In V. W. Marek, A. Nerode, and M. Truszczyński, editors, *Proceedings of the Third International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR '95*, pages 302–315, Lexington, Kentucky, 1995. Springer-Verlag, Lecture Notes in Artificial Intelligence 928. 7, 47

[13] R. F. Stärk. The finite stages of inductive definitions. In P. Hájek, editor, *GÖDEL'96. Logical Foundations of Mathematics, Computer Science and Physics — Kurt Gödel's Legacy*, pages 267–290, Brno, Czech Republic, 1996. Springer-Verlag, Lecture Notes in Logic 6. 7, 59

[14] R. F. Stärk. Total correctness of pure Prolog programs: A formal approach. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proceedings of the 5th International Workshop on Extensions of Logic Programming, ELP '96*, pages 237–254, Leipzig, Germany, 1996. Springer-Verlag, Lecture Notes in Artificial Intelligence 1050. 7

[15] R. F. Stärk. Formal verification of logic programs: foundations and implementation. In S. Adian and A. Nerode, editors, *Logical Foundations of Computer Science LFCS '97 — Logic at Yaroslavl*, pages 354–368. Springer-Verlag, Lecture Notes in Computer Science 1234, 1997. 7

[16] R. F. Stärk. The theoretical foundations of LPTP (a logic program theorem prover). *J. of Logic Programming*, 36(3):241–269, 1998. 7, 31, 40, 86, 87, 96, 102

[17] G. Takeuti. *Proof Theory.* North-Holland, Amsterdam, 1987. 7

# Index

# Index