A TUTORIAL FOR MINLOG, VERSION 5.0

LAURA CROSILLA, MONIKA SEISENBERGER, HELMUT SCHWICHTENBERG

1. Introduction

This is a tutorial for the interactive proof system Minlog, Version 5.0, developed by Helmut Schwichtenberg and members of the logic group[1] at the University of Munich.

Minlog is implemented in Scheme. Minlog's favorite dialect is Petite Chez Scheme from Cadence Research Systems, which is freely distributed at the Internet address `www.scheme.com`.

The Minlog system can be downloaded from the Internet address `http://www.minlog-system.de`

2. Getting started

The purpose of this Tutorial is to give a rather basic introduction to the Minlog system by means of some simple examples. For a thorough presentation of Minlog and the motivation behind it the reader should consult the reference manual [10] and the document [9]. For a more in–depth presentation of the theory underlying Minlog, the reader might find it useful also to consult the book [11]. The papers listed in the Minlog web page also provide a more detailed and advanced description of specific features of the system. In addition, the Minlog distribution comes equipped with a directory of examples, to which the user is referred.

In the following we shall assume tacitly that you are using an UNIX-like operating system and that Minlog is installed in ~/minlog, where ~ denotes as usual your home directory. Also, C-$\langle chr \rangle$ means: hold the CONTROL key down while typing the character $\langle chr \rangle$, while M-$\langle chr \rangle$ means: hold the META (or EDIT or ESC or ALT) key down while typing $\langle chr \rangle$.

Date: May 26, 2020.

This tutorial extends and completes a previous tutorial by L. Crosilla distributed with version 4.0 of Minlog.

[1] http://www.math.lmu.de/∼logik/welcome.html

In order to use Minlog, one simply needs a shell in which to run Minlog and also an editor in which to edit and record the commands for later sessions. In this tutorial we shall refer to GNU Emacs[2]. While working with Emacs, the ideal would be to split the window in two parts: one containing the file in which to store the commands, and the other with the Minlog interactive session taking place. To this aim, it is recommended to use the startup script *~/minlog/minlog* which takes scheme files as (optional) arguments. For example

```
~/minlog/minlog file.scm
```

opens a new Emacs-window which is split into two parts. The upper part contains the file *(Buffer file.scm)* whereas the lower part shows the Minlog response *(Buffer *minlog*)*.

Alternatively, one can open emacs and invoke Minlog by loading the file *minlog.el*:

```
M-x load-file <enter>
~/minlog/minlog.el
```

In both cases the file *init.scm* is loaded. In fact, one could also simply evaluate `(load "~/minlog/init.scm")` to start Minlog.

To execute a command of our file, we simply place the cursor at the end of it (after the closed parenthesis), and type `C-x C-e`. In general, `C-x C-e` will enable us to process any command we type in `file.scm`, one at the time. To process a whole series of commands, one can highlight the region of interest and type `C-c C-r`. We should also mention at this point that to undo one step, it is enough to give the command `(undo)`, while `(undo n)` will undo the last n steps. Finally, we can type `(exit)` to end a Scheme session and `C-x C-c` to exit Emacs.

3. Propositional logic

3.1. A first example.

We shall start from a simple example in propositional logic. Suppose we want to prove the tautology:

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C)).$$

In the following we shall make use of the convention for which parenthesis are associated to the right (as this is also implemented in Minlog). Therefore the formula above becomes:

$$(A \to B \to C) \to (A \to B) \to A \to C.$$

[2] See also the Appendix B.1 for some useful keyboard commands to start working with Emacs.

It is very important, especially at the beginning, to pay the maximum attention to the use of parenthesis to prevent mistakes; it might be a good strategy to rather exceed in their use in the first examples. Minlog will automatically delete the parenthesis which are not needed, therefore facilitating the reading.

3.1.1. *Making a sketch of the proof.* The first task will be to make an informal sketch of the proof. While making the plan we should consider the following fact: Minlog (mainly) implements “Goal Driven Reasoning”, also called “Backward Chaining”. That means that we start by writing the conclusion we aim at as our goal and then, step by step, refine this goal by applying to it appropriate logical rules. A logical rule will have the effect of reducing the proof of a formula, the goal, to the proof of one or more other formulas, which will become the new goals. If our proof is correct, then the formula will be proved when we reach the point of having no more goals to solve. In other words, Minlog keeps a list of goals and updates it each time a logical rule is applied. The proof is completed when the list of goals is empty.

In this case the tautology we want to prove is made of a series of implications, hence we will have to make repeated use of basic rules for “deconstructing” implications. The first move will then be to assume that the antecedent of the outmost implication is true and try to derive the consequent from it. That is, we assume $A \to B \to C$ and want to derive:

$$(A \to B) \to A \to C;$$

hence we set the latter as our new goal. Then we observe that the formula $(A \to B) \to A \to C$ is an implication as well, and thus can be treated in the same way; so we now assume both $A \to B \to C$ and $A \to B$ and wish to derive $A \to C$. Clearly, we can make the same step once more and obtain $A \to B \to C$, $A \to B$ and A as our premises and try to derive C from them. Now we observe that in order to prove C from the assumption $A \to B \to C$, we need to prove both A and B. Obviously A is proved, as it is one of our assumptions, and B immediately follows from $A \to B$ and A by modus ponens.

3.1.2. *Implementing the proof.* Once we have a plan for the proof, we can start implementing it in Minlog. The initial step would then be to write the formula in Minlog. For this purpose, we declare three predicate variables A, B and C by writing [3]:

[3] We could as well have introduced predicate constants instead of predicate variables. In this case we would have used the command `add-predconst-name`, with the same syntax.

```
(add-pvar-name "A" "B" "C" (make-arity))
```

The expression `(make-arity)` produces the empty arity for A, B and C (see [10] for a description of `make-arity`). Minlog will then write:

```
ok, predicate variable A: (arity) added
ok, predicate variable B: (arity) added
ok, predicate variable C: (arity) added
```

We now want to prove the above formula with Minlog; we thus need to set it as our goal.

3.1.3. *Setting the goal.* To set a goal, we can use the command `set-goal` followed by the formula. In the present case:

```
(set-goal "(A -> B -> C) -> (A -> B) -> A -> C")
```

Alternatively, we can first give a name to the formula we wish to prove and then use this name to set the goal. In the present case, let's call `distr` (for distributivity of implication) the formula to be proved:

```
(define distr (pf "(A -> B -> C) -> (A -> B) -> A -> C"))
```

The `define` command has the effect of defining a new variable, in this case `distr`, and attaching to it the Scheme term which is produced by the function `pf` applied to the formula we enter. In fact, the function `pf`, short for "parse formula", takes a string as argument and returns a Scheme term. This Scheme term is the *internal form* in Minlog of our formula, and `distr` is a name referring to it. By typing `distr`, one can see the value of this variable. The strategy of naming a formula might turn out to be particularly useful in case of very long goals.

To set `distr` as our goal we type:

```
(set-goal distr)
```

Typically, Minlog will number the goals occurring in the proof, and will display the top goal as number 1, preceded by a question mark. Minlog will print:

```
?_1: (A -> B -> C) -> (A -> B) -> A -> C
```

3.1.4. *The proof.* According to our sketch, the first step in proving the tautology was to assume the antecedent of the implication and turn the consequent into our new goal. This is simply done by writing:

```
(assume 1)
```

Here the number `1` is introduced by us to identify and name the hypothesis. Minlog will thus denote this hypothesis by `1`:

```
ok, we now have the new goal
?_2: (A -> B) -> A -> C from
  1:A -> B -> C
```

We repeat the assume command to decompose the implication in the second goal and obtain a new goal:

```
(assume 2)
```

```
ok, we now have the new goal
?_3: A -> C from
  1:A -> B -> C
  2:A -> B
```

And we decompose the new goal once more:

```
(assume 3)
```

```
ok, we now have the new goal
?_4: C from
  1:A -> B -> C
  2:A -> B
  3:A
```

We now need to start using our assumptions. As already mentioned, in order to prove C it is enough to prove both A and B, by assumption `1`. Therefore we write: `(use 1)`. This has the effect of splitting the goal in two distinct subgoals (note how the subgoals are numbered):

```
(use 1)
ok, ?_4 can be obtained from
?_6: B from
  1:A -> B -> C
  2:A -> B
  3:A
?_5: A from
  1:A -> B -> C
  2:A -> B
  3:A
```

Then we write:

```
 (use 3)
ok, ?_5 is proved.  The active goal now is
?_6: B from
  1:A -> B -> C
  2:A -> B
  3:A
```

And conclude the proof by:

```
(use 2)
ok, ?_6 can be obtained from
?_7: A from
  1:A -> B -> C
  2:A -> B
  3:A
>
(use 3)
ok, ?_7 is proved.  Proof finished.
```

To see a record of the complete proof, simply type (display-proof). Other useful commands are (proof-to-expr) and the particularly useful (proof-to-expr-with-formulas). See the manual for a description of the various display commands available in Minlog.

We observe that the first three assume commands could be replaced by a single one, i.e., (assume 1 2 3). Also, in alternative to the last two use commands, we could have given only one command: (use-with 2 3), which amounts to applying a cut to the premises 2 and 3. A final remark: in case of rather complex proofs, it may be more convenient to use names to denote specific hypothesis, instead of making use of bare numbers. To do so, one can simply use the assume command, followed by the name of the assumption in double quotes.

Before starting to read the next section it is advisable to consult the reference manual [10] for a compendium of the commands utilized in this example. It is worth noticing that in general these commands have a wider applicability than their usage as here presented.

3.2. **A second example: conjunction.** The next example is a simple tautology made of conjunctions and an implication. We want to prove[4]:

$$A \wedge B \to B \wedge A.$$

In this case, we shall simply record the code of our Minlog proof, asking the reader to check Minlog's reply at each step. We start as usual with declaring the variables A and B and setting the goal. Note that if one uses the same file for a number of examples, there is no need to re-declare the same predicate variables each time. Hence here and in the example file available with the distribution repeated declarations are commented by prefixing a ";; ".

```
;; (add-pvar-name "A" "B" (make-arity))
(set-goal "A & B -> B & A")
```

We than notice that the main connective is an implication, and thus call the command assume:

```
(assume 1)
```

Next we need a command which operates on a conjunction and splits it into its two components. We can simply write:

[4] Recall that ∧ binds stronger than →.

```
(split)
```

Finally, we impart the command `use` which is utilized to obtain the left (respectively the right) conjunct from an assumption which is a conjunction and "`use`" it to derive the goal.

```
(use 1)
(use 1)
```

This completes the proof.

3.3. **Exercises.** The reader is encouraged to try and prove other tautologies. For example the following:

(1) $A \to B \to A$
(2) $(A \to B \to C) \to B \to A \to C$
(3) $(A \to B) \to (B \to C) \to A \to C$
(4) $(A \to B \to C) \to A \wedge B \to C$

3.4. **Classical logic.** To conclude this section on propositional logic, we give a short example of a tautology which uses classical logic.

Minlog implements minimal logic. However, it is possible to use Minlog to prove a proposition which holds in an extension of minimal logic, like intuitionistic or classical logic. This is achieved by adding specific principles which are characteristics of each kind of logic, like the "ex falso quodlibet" or a version of the "tertium non datur". We recall that intuitionistic logic is obtained from minimal logic by adding a principle which enables us to derive any formula from falsity ("ex falso quodlibet" is Latin for "anything follows from falsity"). This principle is usually stated as: $\bot \to A$, for arbitrary A, where $\bot$ denotes falsity. Classical logic may be obtained by adding to intuitionistic logic the law of "tertium non datur" (Latin for "there is no third option"). This is usually stated as $A \vee \neg A$, for any A. However, classical logic may also be obtained by adding to minimal logic a consequence of "tertium non datur", known as "stability", asserting that $\neg\neg A \to A$, for any A. Negation is represented in Minlog as follows: $\neg A$ is $A \to \bot$; consequently stability is written as:

$$((A \to \bot) \to \bot) \to A,$$

for each A.

We can add "external" principles to Minlog by introducing so–called *"global assumptions"*. Roughly speaking, a global assumption is a proposition whose proof does not concern us at the moment; hence it can also be an assumption with no proof. Some global assumptions are already set by default, like `EfqLog` (which is Minlog's name for "ex falso quodlibet") and `StabLog` (which is Minlog's name for the law of stability). In order to check which global assumptions we have at our disposal we type: `(display-global-assumptions)`. To check a particular global assumption (or theorem) whose name we already know, we write `pp` (for pretty-print) followed by the name of the assumption (or theorem) we want to check, e.g.: `(pp "StabLog")`. Of course we can also introduce our own global assumptions and remove them at any time (see the reference manual for the relevant commands).

In the following we wish to prove the tautology:

$$((A \to B) \to A) \to A,$$

which is known as *Peirce formula*. Also for this example, we will assume that the reader has prepared her sketch of the proof, and we will only give an intuitive idea of the proof, preferring to rather concentrate on the Minlog interaction, which will be given in its complete form.

As in the previous examples, we observe first of all that the goal is an implication, hence we will assume its antecedent, $(A \to B) \to A$, and try to prove its consequent, A. Now classical logic comes into play: in order to prove A we will assume that its negation holds and try to obtain a contradiction from it. This will be achieved by use of Stability. We further note that in order to make the argument work, we will need at some stage to resort also to “ex falso quodlibet”.

We start by setting the goal and assuming the antecedent of the implication:

```
;; (add-pvar-name "A" "B" (make-arity))
(set-goal "((A -> B) -> A) -> A")
(assume 1)
```

We obtain:

```
ok, we now have the new goal
?_2: A from
  1:(A -> B) -> A
```

We now apply Stability, `StabLog`, so that the goal A will be replaced by its double negation: $(A \to \bot) \to \bot$. Note that $\bot$ is called `bot` in Minlog.

```
(use "StabLog")
ok, ?_2 can be obtained from
?_3: (A -> bot) -> bot from
  1:(A -> B) -> A
```

Since this is an implication, we let:

```
(assume 2)
ok, we now have the new goal
?_4: bot from
```

```
  1:(A -> B) -> A
  2:A -> bot
```

We then use hypothesis 2 to replace the goal $\bot$ by A.

```
(use 2)
ok, ?_4 can be obtained from
?_5: A from
  1:(A -> B) -> A
  2:A -> bot
```

Also A can be replaced by $A \to B$ by use of hypothesis 1. Subsequently, we can assume the antecedent of the new goal, A, and call it hypothesis 3:

```
(use 1)
ok, ?_5 can be obtained from
?_6: A -> B from
  1:(A -> B) -> A
  2:A -> bot
>
(assume 3)
ok, we now have the new goal
?_7: B from
  1:(A -> B) -> A
  2:A -> bot
  3:A
```

Now we can make use of the principle of “ex falso quodlibet”: if we want to prove B, we can instead prove falsum, since from falsum anything follows, in particular B. Our goal can be updated to $\bot$ by the following instance of `use`:

```
(use "EfqLog")
ok, ?_7 can be obtained from:
?_8: bot from
  1:(A -> B) -> A
  2:A -> bot
  3:A
```

The next two steps are obvious.

```
(use 2)
ok, ?_8 can be obtained from
?_9: A from
  1:(A -> B) -> A
  2:A -> bot
  3:A
```

```
>
(use 3)
ok, ?_9 is proved.  Proof finished.
```

3.5. **Exercises.** To familiarize yourself with negation, prove the following propositions.

(1) $(A \to B) \to \neg B \to \neg A$,
(2) $\neg(A \to B) \to \neg B$,
(3) $\neg\neg(A \to B) \to \neg\neg A \to \neg\neg B$.

3.6. **Note on disjunction.** In Minlog we don't have a primitive logical constant for $\vee$. This is due to the fact that when wishing to prove the disjunction $A \vee B$ we can often prove

```
(A -> Pvar) -> (B -> Pvar) -> Pvar
```

instead[5].

4. Predicate logic

4.1. **A first example with quantifiers.** We now exemplify how to prove a statement in predicate logic. We want to prove:

$$\forall_n(Pn \to Qn) \to \forall_n Pn \to \forall_n Qn.$$

Here we assume that the predicates P and Q take natural numbers as arguments. Therefore, we first of all load a file, already available within the distribution, which introduces the algebra of natural numbers, including some operations on them, like for example addition. The reader is advised to have a look at this file[6] by typing:

```
(libload "nat.scm")
```

Note that this command will produce the display of the whole file nat.scm, evaluated. If we wish to load the file "silently", then we can precede the libload command by the following line:

```
(set! COMMENT-FLAG #f)
```

This will have the effect of hiding the output. To revert to full display (which is needed to proceed with the proof) one types:

```
(set! COMMENT-FLAG #t)
```

[5] Note, however, that there is also an (inductively defined) "or", which is displayed as `ord`, with rules:

```
(pp "InlOrD")
Pvar1 -> Pvar1 ord Pvar2
(pp "InrOrD")
Pvar2 -> Pvar1 ord Pvar2
```

[6] See also section 6.1.

As we load `nat.scm`, we can make use of all the conventions which are there stipulated; in particular, we can take n, m, k to be variables for natural numbers (i.e. of type `nat`)[7]. The next task is to introduce two new predicate variables P and Q which take natural numbers as arguments:

```
(add-pvar-name "P" "Q" (make-arity (py "nat")))
> ok, predicate variable P: (arity nat) added
ok, predicate variable Q: (arity nat) added
```

We then set the goal:

```
(set-goal "all n(P n -> Q n) -> all n P n -> all n Q n")
> ?_1: all n(P n -> Q n) -> all n P n -> all n Q n
```

As usual, we first have to "deconstruct" the implications:

```
(assume 1 2)
> ok, we now have the new goal
?_2: all n Q n from
  1:all n(P n -> Q n)
  2:all n P n
```

Then we need to take care of the universal quantifier in the goal:

```
(assume "n")
> ok, we now have the new goal
?_3: Q n from
  1:all n(P n -> Q n)
  2:all n P n
  n
```

Note that we could have also used only one command to perform all these actions:

```
(assume 1 2 "n")
```

We finally have to "use" our hypothesis to conclude the proof:

```
(use 1)
(use 2)
> ok, ?_3 can be obtained from
?_4: P n from
  1:all n(P n -> Q n)
  2:all n P n
  n
> ok, ?_4 is proved.  Proof finished.
```

[7] Note that Minlog automatically infers the type of the variables $x_0, x_1, \ldots$ once the variable x has been declared. In this case, it infers for example that $n_0, n_1, \ldots$ are also natural numbers.

4.2. **Another example.** We now wish to prove the following:

$$\forall_n (Pn \to Qn) \to \exists_n Pn \to \exists_n Qn.$$

We start by setting the goal and eliminating the two implications:

```
(set-goal "all n(P n -> Q n) -> ex n P n -> ex n Q n")
(assume 1 2)
?_1: all n(P n -> Q n) -> ex n P n -> ex n Q n
> ok, we now have the new goal
?_2: ex n Q n from
  1:all n(P n -> Q n)
  2:ex n P n
```

Next we want to use the second assumption, now by applying "forward reasoning". Thus we assume there is a witness, say n_0, for this existential formula and call the resulting hypothesis P_{n_0}. To this aim we make use of a command called `by-assume`:

```
(by-assume 2 "n0" "P_n0")
> ok, we now have the new goal
?_5: ex n Q n from
  1:all n(P n -> Q n)
  n0  P_n0:P n0
```

Alternatively, from the existential formula in hypothesis 2, we can extract a witness, say n_0, by applying an existential elimination.

```
(ex-elim 2)
(assume "n0" "P_n0")
> ok, ?_2 can be obtained from
?_3: all n(P n -> ex n0 Q n0) from
  1:all n(P n -> Q n)
  2:ex n P n
> ok, we now have the new goal
?_4: ex n Q n from
  1:all n(P n -> Q n)
  2:ex n P n
  n0  P_n0:P n0
```

As we have already used assumption 2, we may wish to drop it by writing:

```
(drop 2)
```

We conclude the proof by providing a witness, the term n_0, for our existential goal formula. This will be done by calling the command `ex-intro` with argument `(pt "n0")`, where `pt` stands for "parse term".

```
(ex-intro (pt "n0"))
```

```
> ok, ?_5 can be obtained from
?_6: Q n0 from
  1:all n(P n -> Q n)
  n0  P_n0:P n0
```

Finally, we first instantiate the universal quantifier in assumption 1 to n_0 and then perform a cut with the third assumption.

```
(use-with 1 (pt "n0") "P_n0")
>  ok, ?_5 is proved.  Proof finished.
```

4.3. **An example with relations.** In the next example we wish to prove that every total relation which is symmetric and transitive is reflexive. For simplicity we shall work also in this case with the algebra of the natural numbers. Our aim is to prove the following statement:

$$\forall_{n,m}\,(Rnm \to Rmn) \wedge \forall_{n,m,k}\,(Rnm \;\wedge\; Rmk \to Rnk)$$
$$\to \forall_{n}\,(\exists_{m}\,Rnm \to Rnn),$$

where n, m, k vary on natural numbers, while R is a binary predicate on natural numbers.

Before attacking our formula, we observe that in general conjunctions are quite complex to deal with, as they normally imply the branching of a proof in two subproofs. Thus we might wish to first find a formula which is equivalent to the one above and “simpler” to prove. We note that we can equivalently express our goal by a formula in which the conjunctions have been replaced by implications. Also, we can express the conclusion with a prenex universal quantifier instead of an existential one. That is, we can instead prove the following equivalent formula:

$$\forall_{n,m}\,(Rnm \to Rmn) \to \forall_{n,m,k}\,(Rnm \to Rmk \to Rnk)$$
$$\to \forall_{n,m}\,(Rnm \to Rnn).$$

We observe that the strategy of first simplifying the goal may in some cases allow one to considerably reduce the amount of time needed to prove a statement. For completeness and for a comparison, we shall also record a proof of the original goal at the end of this section.

We now start by introducing the constant R. We also want to facilitate our work a bit further and introduce names for our two assumptions. In the following we shall use the function py (for “parse type”), which is the analogous for types of the function parse formula that we encountered in the first example.

```
(add-pvar-name "R" (make-arity (py "nat") (py "nat")))
(define Sym (pf "all n,m(R n m  -> R m n)"))
```

```
(define Trans (pf "all n,m,k(R n m -> R m k -> R n k)"))
```

We now state the goal:

```
(set-goal (mk-imp Sym Trans (pf "all n,m(R n m -> R n n)")))
?_1: all n,m(R n m -> R m n)
       -> all n,m,k(R n m -> R m k -> R n k)
       -> all n,m(R n m -> R n n)
```

Note that also in this case, we could have directly written the two formulas as antecedents of the implication, avoiding the detour through a `define` command. In case of more complex formulas, however, or when we need to use the same formulas for various proofs through one session, the strategy of introducing names for assumptions can be quite useful.

We now observe that the goal is an implication, so that the first step is to write `(assume "Sym" "Trans")`. We now obtain a universally quantified formula and hence need to proceed to eliminate the quantifiers. This can be accomplished by another `assume` command in which we specify two natural numbers, say n and m. So we write `(assume "n" "m")`. This produces an implication which again needs to be eliminated by another `assume` command, say `(assume 3)`. Quite conveniently we can put all these commands together by simply writing:

```
(assume "Sym" "Trans" "n" "m" 3)
ok, we now have the new goal
?_2: R n n from
  Sym:all n,m(R n m -> R m n)
  Trans:all n,m,k(R n m -> R m k -> R n k)
  n  m  3:R n m
```

The next move is to make use of our assumptions. It is clear that if we take k to be n in `Trans`, then the goal can be obtained by an instance of `Sym`, and the proof is easily completed. We here utilize `use` by additionally providing a term, `"m"`, which instantiates the only variable which can not be automatically inferred by unification[8].

```
(use "Trans" (pt "m"))
?_4: R m n from
  Sym:all n,m (R n m -> R m n)
  Trans:all n,m,k(R n m -> R m k -> R n k)
  n  m  3:R n m
?_3: R n m from
  Sym:all n,m(R n m -> R m n)
```

[8] See [9] for an introduction to unification.

```
  Trans:all n,m,k(R n m -> R m k -> R n k)
  n  m  3:R n m
```

The use command has the effect of replacing the current goal with two new goals. These are obtained from Trans by instantiating the quantifiers with n, m and n (the two n being inferred by unification) and then by replacing the goal with the antecedents of the resulting instance of Trans. We can now write:

```
(use 3)
> ok, ?_3 is proved.  The active goal now is
?_4: R m n from
  Sym:all n,m(R n m -> R m n)
  Trans:all n,m,k(R n m -> R m k -> R n k)
  n  m  3:R n m
```

We finally employ Sym and another use:

```
(use "Sym")
ok, ?_4 can be obtained from
?_5: R n m from
  Sym:all n,m(R n m -> R m n)
  Trans:all n,m,k(R n m -> R m k -> R n k)
  n  m  3:R n m
>
(use 3)
ok, ?_5 is proved.  Proof finished.
```

4.4. **The same example again.** We here present a Minlog proof of the original goal in the previous example, as it allows us to exemplify the use of some new commands. We shall leave the proof uncommented and make a few remarks at the end. The reader will have to examine the proof and check Minlog's interaction.

```
;; (libload "nat.scm")
;; (add-pvar-name "R" (make-arity (py "nat") (py "nat")))
(set-goal "all n,m(R n m  -> R m n)
                 & all n,m,k(R n m & R m k -> R n k)
                 -> all n(ex m R n m -> R n n)")
(assume 1)
(inst-with 1  ’left)
(inst-with 1 ’right)
(drop 1)
(name-hyp 2 "Sym")
(name-hyp 3 "Trans")
(assume "n" 4)
```

```
(ex-elim 4)
(assume "m" 5)
(cut "R m n")
(assume 6)
(use-with "Trans" (pt "n") (pt "m") (pt "n") "?")
(drop "Sym" "Trans" 4)
(split)
(use 5)
(use 6)
(use-with "Sym" (pt "n") (pt "m") 5)
```

The `use-with` command is similar to the `use` command, but when applied to a universal quantifier it requires to explicitly specify the terms one wants to instantiate. In the second occurrence of `use-with`, Minlog will instantiate as specified the universal quantifiers in the second premise and then use hypothesis 5 to prove the goal.

The command `inst-with` is analogous to `use-with`, but operates in forward reasoning; hence it allows one to simplify the hypothesis, instead of the conclusion. In this case, `(inst-with 1 'left)` has the effect of producing the left component of the conjunction which constitutes the first hypothesis. Similarly for the right component.

As to `cut`, this command enables one to introduce new goals: `(cut A)` has the effect of replacing goal B by two new goals, $A \to B$ and A.

In the proof above we have also made use of the commands `drop` and `name-hyp`. We have already seen the first command, which allows one to remove one or more hypothesis from the present context, to make the proof more readable. In fact, it simply replaces the current goal with another goal in which the hypothesis "dropped" are not displayed anymore (but they are not removed in general, as should be clear from the example above). The second command has similar "cosmetic" purposes, and allows one to rename a specific hypothesis and hence to work with names given by the user instead of numbers produced by default. Both these commands result especially useful in the case of long and intricate proofs.

4.5. **Exercises.** Prove the following goals:

(1) $\forall_{m,n} Rmn \to \forall_{n,m} Rmn$
(2) $\forall_{m,n} Rmn \to \forall_{n} Rnn$
(3) $\exists_{m} \forall_{n} Rmn \to \forall_{n} \exists_{m} Rmn$

4.6. **Advanced exercises.** Now two examples which involve a function. First declare a new function variable (where `av` stands for "add variable"):

```
(av "f" (py "nat=>nat"))
(set-goal "all f(all n(P(f n) -> Q n)
-> all n P n -> all n Q n)")
(set-goal "all f(all n(P n -> Q (f n))
-> ex n P n -> ex n Q n)")
```

And finally:

```
;; (add-pvar-name "Q" (make-arity (py "nat")))
;; (add-pvar-name "A" (make-arity))
(set-goal "all n(Q n -> A) -> (ex n Q n -> A)")
(set-goal "ex n(Q n -> A) -> all n Q n -> A")
```

4.7. **Another example with classical logic.** We conclude this section on predicate logic with the proofs of two formulas which hold in classical logic. First of all, we prove the inverse of the formula in the last exercise, now generalised to an arbitrary type. Then we use this formula (conveniently stored as a Lemma) to prove another formula which is usually known as the "Drinker" formula. So we start by proving:

$$(\forall_x Qx \to A) \to \tilde{\exists}_x (Qx \to A).$$

Here Q is a unary predicate which ranges on an arbitrary type, say α. In addition, the existential quantifier, $\tilde{\exists}$, is here a **classical** existential quantifier, to be distinguished from the existential quantifier we encountered in the previous example. A classical quantifier $\tilde{\exists}_x$ is nothing more than an abbreviation for $\neg \forall_x \neg$. Note that Minlog implements both quantifiers, with the appropriate corresponding rules.

We start by "removing" the constant Q from example 4.1, so that we can re-introduce it as a fresh constant which ranges on α rather than on the natural numbers. We also introduce two new variables, x and y, of type α. Finally, we set the goal.

```
(remove-pvar-name "Q")
(add-pvar-name "Q" (make-arity (py "alpha")))
(av "x" (py "alpha"))
(set-goal "(all x Q x -> A) -> excl x(Q x -> A)")
```

We start by "deconstructing" the two implications. Then we can instantiate the universal quantifier in assumption 2 to a canonical inhabitant of the type α.

```
(assume 1 2)
(use 2 (pt "(Inhab alpha)"))
```

Subsequently, we proceed by eliminating the implication in the goal, using assumption 1 and instantiating the resulting universal quantifier by x.

```
(assume 3)
(use 1)
(assume "x")
```

Now it's time to call in classical logic. The remaining steps should be self–explanatory. Note in particular the use of `EfqLog` and the command `save` at the end of the proof which enables us to save the proof and call it "Lemma".

```
(use "StabLog")
(assume 4)
(use 2 (pt "x"))
(assume 5)
(use 1)
(assume "x1")
(use "EfqLog")
(use-with 4 5)
(save "Lemma")
```

We now wish to prove the following:

$$\tilde{\exists}_x \, (Qx \to \forall_y \, Qy),$$

again with Q a unary predicate ranging on α.

The above formula is known as the "drinker" formula, as it says something like: "in a bar, there is a person such that if that person drinks then everybody drinks". To prove the "drinker", we observe that if we substitute the predicate variable A by $\forall_x Qx$ in the formula just proved, then we obtain the drinker formula. The substitution can be achieved by the following command.

```
(set-goal "excl x(Q x -> all x Q x)")
(use-with "Lemma"
          (make-cterm (pv "x") (pf "Q x"))
          (make-cterm (pf "all x Q x"))
          "?")
```

Here `pv` stands for "parse variable". In addition, `make-cterm` produces a *"comprehension term"* consisting of a list of variables and the formula we wish to substitute. For example, if we wish to replace a predicate variable P with arity $x_1, \dots, x_n$ by a formula $F(y_1, \dots, y_n)$ we need to give a comprehension term consisting of a list of variables $y_1, \dots, y_n$ and the formula F (with free variables $y_1, \dots, y_n$, plus possibly other variables, bound or free). That is, we write: `(make-cterm (pv "y1") ...(pv "yn") <formula F> )`. Note that the list of variables can also be empty, as in the second application of `make-cterm` above. We leave the rest of the proof as an exercise for the reader.

4.8. Equality reasoning.

We now wish to prove that for any function f taking natural numbers to natural numbers, for any natural number n, the following holds:

$$fn = n \to f(fn) = n.$$

First of all we recall the nat library and introduce f. Then we set the goal and start by a familiar `assume` command.

```
;; (libload "nat.scm")
(av "f" (py "nat=>nat"))

(set-goal "all f,n(f n=n ->  f(f n)=n)")
(assume "f" "n" 1)
```

Next we can use the command `simp` which is an essential tool in Minlog's equality reasoning. This command has the effect of *simplifying* a proof which involves equal terms by performing an appropriate substitution in the goal. We conclude with a `use` command.

```
(simp 1)
(use 1)
```

Suppose now we wish to replace the right hand side by the left hand side in the equation above:

```
(set-goal "all f,n(n=f n -> n=f(f n))")
```

This can be proved by the following commands:

```
(assume "f" "n" 1)
(simp "<-" 1)
(use 1)
```

5. Automatic proof search

Minlog allows for automatic proof search. There are two distinct facilities for performing an automatic search in Minlog. The first is given by the command `(prop)` and exemplifies Hudelmaier-Dyckhoff's search for the case of minimal propositional logic (see e.g. [5], [4]). The second is given by the command `(search)` and allows to automatically find proofs also for some quantified formulas.

5.1. Search in propositional logic.

When we give the command `prop`, Minlog will first look for a proof in propositional minimal logic. If it fails to find a proof for the given proposition, it will try with intuitionistic logic, by adding appropriate instances of “ex falso quodlibet”. If this search also gives no positive answer, it will try to find a proof in classical logic, by adding appropriate instances of Stability.

To apply this search algorithm, one simply needs to type `(prop)`. One could do so after stating the goal or at any point in a proof from which one believes that (minimal) propositional logic should suffice. If Minlog finds a proof, one can then display it by means of any of the display commands available for proofs; for example by writing `dnp` (which is a shortcut for `display-normalized-proof`).

The reader is encouraged to try `prop` on the following tautologies:

(1) $(A \to B \to C) \to (A \to B) \to A \to C$
(2) $((A \to B) \to A) \to A$

Further test examples can be found in the section on propositional logic in this tutorial.

5.2. **Search in predicate logic.** The command `search` embodies a search algorithm based on [7] and ideas of U. Berger (see the Minlog reference manual and [8] for details on the algorithm and for some differences with Miller's original algorithm). The `search` command enables us to automatically find a proof for a wider class of formulas compared with `prop`, since it also works for some formulae with quantifiers (see the reference manual for a detailed description of the class of formulae dealt with by `search`). Note, however, that `search` only operates a search in *minimal logic*. If one wishes to apply this command to a classical formula like "Peirce's law", one could for example add the appropriate instances of "ex falso quodlibet" and of "Stability" as antecedents of the goal. In case of more complex proofs, in which one can not easily modify the actual goal, an alternative would be to avail oneself of a more complete use of the `search` command which allows us to specify some global assumptions, theorems or even hypotheses from the given context which one would like to use in the proof. Since the search space in the case of quantified formulas can become really vast, this possibility of declaring specific assumptions to be used in the proof can be very useful, especially if we also state the maximum number of multiplicities we allow for each assumption (i.e., the maximum number of times each assumption can be used in the proof). One can also use this same device to exclude the use of a specific assumption in the proof, simply by letting its multiplicity to be 0.

To use the plain version of `search`, one simply writes `(search)`. See the reference manual for the precise syntax of the command `search` when other assumptions are invoked with the respective multiplicities.

The reader is encouraged to use `search` to prove the following: $\forall_x (Px \to Qx) \to \exists_x Px \to \exists_x Qx$.

5.2.1. *A more complex example with* **`search`**. We here wish to introduce a more complex example for the use of `search`. We apply the algorithm to the following problem: if f is a continuous function then f composed with itself is also a continuous function. We suggest to solve the problem as follows.

```
;; (add-var-name "x" "y" (py "alpha"))
(add-tvar-name "beta")
(add-var-name "u" "v" "w" (py "beta"))
(add-infix-display-string "In" "elem" 'rel-op)
(add-var-name "f" (py "alpha=>alpha"))

(set-goal "all f(
 all x,v(f x elem v ->
  excl u(x elem u & all y(y elem u -> f y elem v))) ->
 all x,w(f(f x)elem w ->
  excl u(x elem u & all y(y elem u -> f(f y)elem w))))")
(search)
```

Note that one can switch on a verbose form of search by letting: `(set! VERBOSE-SEARCH #t)` before calling `search`. In this way one can see the single steps performed by the search algorithm and detect possible difficulties in finding a proof.

Also, `add-infix-display-string` allows us to define a token with infix notation for the program constant[9].

6. Datatypes and inductively defined predicates

At this point we prefer to slightly change our style how goals appear on the screen. The reason is that from now on our proofs become somewhat more complex, and then the new style is more readable.

- For somewhat longer contexts it is clearer to list them first and the goal below it, separated by a line. The command to switch to this display style is

  ```
  (set! COQ-GOAL-DISPLAY #t)
  ```

- We avoid using numbers to identify hypotheses and rather use strings indicating what kind of a hypothesis we have. For example “IHn” tells us that this is the hypothesis of an induction over n.

[9] Similarly there are commands for prefix and postfix use, for example: `add-prefix-display-string`.

6.1. **The natural numbers.** The standard example of a datatype is that of the natural numbers. We have already seen that the natural numbers are implemented in Minlog as an algebra, and that the distribution comes equipped with a file, called `nat.scm`, which introduces this algebra. The algebra's constructors are 0 and *Succ* (zero and successor). To display these constructors, we simply write:

```
(display-alg "nat")
```

We obtain Minlog's reply:

```
> nat
        Zero:         nat
        Succ:         nat=>nat
```

Note also that for convenience Minlog allows us to write `0, 1, 2, 3, ...` instead of `Zero, (Succ Zero), Succ(Succ Zero), ....`

Algebras usually come equipped with some functions, which are called **program constants** in Minlog. For example, in the case of the natural numbers, one has the program-constants `NatPlus` and `NatTimes`, for addition and multiplication, respectively. These are displayed as `+` and `*`. The behaviour of program constants is specified by means of appropriate term rewriting rules which in Minlog are called **computation rules** and **rewrite rules** [10].

For example, to see the program constant `NatPlus` and its rules type:

```
(display-pconst "NatPlus")

> NatPlus
  comprules
        nat+0         nat
        nat1+Succ nat2        Succ(nat1+nat2)
  rewrules
        0+nat         nat
        Succ nat1+nat2        Succ(nat1+nat2)
        nat1+(nat2+nat3)      nat1+nat2+nat3
```

Note that here `nat` is a default variable of type nat. We recommend to have a look at the file `nat.scm` to familiarise oneself with the way program constants are defined.

To see the effect of term rewriting rules for `+` we type

```
(pp (nt (pt "3+4")))
```

[10] The idea is that a computation rule can be understood as a description of a computation in a suitable *semantical* model, provided the syntactic constructors correspond to semantic ones in the model, whereas the other rules should be proved before being introduced.

```
(pp (nt (pt "Succ n+Succ m+0")))
```

which yields as results the number `7` and `Succ(Succ(n+m))`. Here `pp` stands for `pretty print` and `nt` stands for `normalize term`; this essentially consists in repeatedly applying[11] the term rewriting rules until no new term is obtained.

6.1.1. *Adding new program constants and computation rules.* We now wish to exemplify the introduction of new program constants on the natural numbers.

Recall that if the file `nat.scm` is not already loaded[12], we can type:

```
(set! COMMENT-FLAG #f)
(libload "nat.scm")
(set! COMMENT-FLAG #t)
```

We now introduce a new program constant which represents the function which doubles a natural number. The command used to introduce a new program constant is `add-program-constant`. It requires the name of the constant and its type; further arguments may be the degree of totality, the token type (e.g. `const`) and the arity (see [10]). In particular, note that in Minlog we can treat not only total objects but also partial ones[13]. Therefore, when we introduce a new program constant, we may also specify its totality degree. A totality degree of one (`t-deg-one`) indicates that the program constant is total, while zero (which is the default) denotes non–totality. As to the type, in the present case, the new constant `Double` is of arrow type, as it takes natural numbers as input and produces natural numbers as output.

```
(add-program-constant "Double" (py "nat=>nat"))
```

In case we wish to remove this program constant, we simply write:

```
(remove-program-constant "Double")
```

The behaviour of a new program constant can be specified by introducing one or more computation rules for it. This is accomplished by use of the command `add-computation-rule`, having two arguments: a left hand side and a right hand side. The right hand side specifies the result of the computation rule for the argument indicated in the left hand side.

The following example should clarify how to use these commands.

[11] Term rewriting in Minlog makes use of normalisation-by-evaluation (see [10]).

[12] Clearly, it is good practice to run a new Minlog session when loading new files which could turn out to be incompatible with previously loaded files or previously introduced definitions.

[13] For the notion of totality see [12, Chapter 8.3]; see also [11].

The function "Double" can be defined by specifying primitively recursively how it acts on zero and on the successor of each natural number.

```
(add-computation-rule (pt "Double 0") (pt "0"))
(add-computation-rule (pt "Double(Succ n)")
                      (pt "Succ(Succ(Double n))"))
```

Alternatively, one could also write:

```
(add-computation-rules
 "Double 0" "0"
 "Double(Succ n)" "Succ(Succ(Double n))")
```

To see the effect of the newly introduced computation rules:

```
(pp (nt (pt "Double 3")))
(pp (nt (pt "Double(n+2)")))
```

6.1.2. *Proof by induction.* Here we wish to exemplify a proof by induction on the natural numbers. The goal is very simple: we wish to show that `Double` $n = n + n$. The first step of the proof consists in using the command `ind`. This command requires a universally quantified goal and proves it by induction, according to the definition of the specific algebra type.

```
(set-goal "all n Double n=n+n")
(ind)
```

The effect of applying `ind` is to refine the goal to a proof of the base and the step cases of the induction. In the present case, where the constructors are Zero and Successor, we have to prove two cases: one for Zero and one for Successor. Minlog's reply will be something like this:

```
ok, ?_1 can be obtained from

  n2218
------------------------------------------------------
?_3:all n(Double n=n+n -> Double(Succ n)=Succ n+Succ n)

  n2218
------------------------------------------------------
?_2:Double 0=0+0
```

We then replace the goal with its normal form by letting:

```
(normalize-goal)

  n2218
```

```
--------------------------------------------------------
?_4:T
```

The latter command can be abbreviated with `ng` and it normalizes the goal by using the computation rules for "+" introduced in the file `nat.scm`. More specifically, as both Double 0 and 0+0 reduce to 0, the normalization will first of all produce 0=0. This in turn reduces to truth, here indicated by `T`. It comes equipped with an axiom `Truth`, by means of which we prove the base case.

```
(use "Truth")

ok, ?_4 is proved.  The active goal now is
  n2218
--------------------------------------------------------
?_3:all n(Double n=n+n -> Double(Succ n)=Succ n+Succ n)
```

As to the step, we make use of the induction hypothesis, `IH`, and write:

```
(assume "n" "IH")
(ng)
(use "IH")

> ok, we now have the new goal
  n2218  n  IH:Double n=n+n
--------------------------------------------------------
?_5:Double(Succ n)=Succ n+Succ n
> ok, the normalized goal is
  n2218  n  IH:Double n=n+n
--------------------------------------------------------
?_6:Double n=n+n
> ok, ?_6 is proved.  Proof finished.
```

Also in this case, when we write `ng` the term rewriting rules for `Double` and "+" are applied.

Finally, we wish to recall that one could also define the Double function without making use of a primitive recursive definition.

```
(add-program-constant "DoubleN" (py "nat=>nat"))
(add-computation-rule (pt "DoubleN n") (pt "n+n"))
```

6.1.3. *Exercises.* Prove that the two definitions of the doubling function are equivalent:

```
(set-goal "all n Double n=DoubleN n")
```

Prove also the following:

```
(set-goal "all n,m n+m=m+n")
```

6.1.4. *Rewrite rules.* Once we have proved the above statement for which the two definitions of Double are equivalent, we may add a rewrite rule which replaces each occurrence of `Double` by `DoubleN`.

```
(add-rewrite-rule (pt "Double n") (pt "DoubleN n"))
```

6.1.5. *Another example.* We now present another example of induction on the natural numbers, which introduces some additional features of Minlog.

Suppose we want to prove that for all natural numbers n, *Double* n is even. We define two new program constants `Odd` and `Even` which take a natural number as argument and give a boolean (true or false) as output. As usual, the behaviour of these program constants can be specified by means of appropriate computation rules. In this case the computation rules will simultaneously characterize `Odd` and `Even`.

```
(add-program-constant "Odd" (py "nat=>boole"))
(add-program-constant "Even" (py "nat=>boole"))

(add-computation-rules
 "Odd 0" "False"
 "Even 0" "True"
 "Odd(Succ n)" "Even n"
 "Even(Succ n)" "Odd n")
```

The steps of the proof are self–explanatory:

```
(set-goal "all n Even(Double n)")
(ind)
(prop)
(search)
```

6.2. **Case distinction on the booleans.** We wish to give an example of distinction by cases, and for simplicity we shall consider a trivial example on the booleans.

We wish to prove that for any boolean p, if p is not false then it is true. We add a variable `p` of type boolean and set the goal:

```
(av "p" (py "boole"))
(set-goal "all p((p=False -> F) -> p=True)")
```

The proof then proceeds by cases: either p is false or it is true. The following steps should be clear.

```
(cases)
(prop)
(prop)
```

6.3. **Induction on lists.**

The following example is an exercise on lists over an arbitrary type α. This example illustrates again the use of induction; however, since we now deal with *parametrized algebras* (see [10, 11]) the task turns out to be a bit harder than when working with the algebra of natural numbers.

To start with we load the file `list.scm`, which contains basic definitions and operations on lists over an arbitrary type α[14]. Then we introduce a function, Rv, on lists which has the effect of reverting a list. Finally we prove:

$$\forall_{v,w}(\mathrm{Rv}\,(v * w) \equiv (\mathrm{Rv}\,w) * (\mathrm{Rv}\,v)),$$

where v and w are lists over an arbitrary type α and $*$ denotes the append function on lists as defined in `list.scm`. Further, $\equiv$ represent Leibniz' equality[15]: two elements are equal if they have the same properties, i.e., they are indistinguishable.

We begin as follows:

```
;; (libload "nat.scm")
(set! COMMENT-FLAG #f)
(libload "list.scm")
(set! COMMENT-FLAG #t)

(add-var-name "x" "a" "b" "c" "d" (py "alpha"))
(add-var-name "xs" "v" "w" "u" (py "list alpha"))
```

We now need to define Rv. This is defined inductively, by first giving its value for the empty list and then saying how it applies to a nonempty list. The two defining conditions for Rv are the following:

$$\mathrm{Rv}\,(\mathrm{Nil}\,\alpha) = (\mathrm{Nil}\,\alpha),$$
$$\mathrm{Rv}\,(a :: w) = (\mathrm{Rv}\,w) * (a{:})$$

where, according to the notation in `list.scm`, $(\mathrm{Nil}\,\alpha)$ denotes the empty list over the type α, $a :: w$ denotes the list obtained by adding the object a of type α to the list w (over α), while a: is the one element list obtained from a. We thus write:

```
(add-program-constant "ListRv"
            (py "list alpha => list alpha") t-deg-one)
(add-prefix-display-string "ListRv" "Rv")
```

[14] Note that `list.scm` does require to first upload `nat.scm`. We recommend to go through the list file before working out this example.

[15] Internally Leibniz equality is printed `eqd`, where the `d` stands for "defined", since Leibniz equality is inductively defined by the clause `InitEqD`: $\forall^{\mathrm{nc}}_x x \equiv x$; see 6.5 for inductively defined predicates.

```
(add-computation-rules
 "Rv(Nil alpha)" "(Nil alpha)"
 "Rv(x::xs)" "Rv xs++x:")
```

Note that for simplicity we have stated that `ListRv` is a total function. Minlog's output will include a warning, to remind us that we should have separately proved before that `ListRv` is in fact total. Also, `add-prefix-display-string` allows us to define a token for the program constant[16].

The following proof makes use of a program constant, `ListAppd`, already available within the file `list.scm`. This has the following computation rules:

```
(Nil alpha)++xs2      xs2
(x1::xs1)++xs2        x1::xs1++xs2
```

And rewrite rules:

```
xs++(Nil alpha)       xs
xs1++x2: ++xs2        xs1++(x2::xs2)
```

To check `ListAppd` we type:

```
(display-pconst "ListAppd")
```

Now we can set the goal and start the proof by calling `ind`:

```
(set-goal "all v,w Rv(v++w)eqd Rv w++Rv v")
(ind)
```

This has the effect of producing two subgoals, corresponding to the base case and the step case, respectively. We tackle the base case as follows:

```
(ng)
(assume "w")
(use "InitEqD")
```

Here we have used `InitEqD`, which is the axiom: `xs eqd xs`.

Subsequently we move to the step case:

```
(assume "a" "v" "IHw" "w")
(ng)
(simp "IHw")
```

And finally we use a theorem proved in the file `list.scm` and there called `ListAppdAssoc`:

[16] In the file `list.scm` there already exists a program constant `ListRev` with display string `Rev` defined exactly as our `ListRv`. Here we have just duplicated the definition for pedagogical reasons, since it is a nice and easy example. However, we had to choose a different name to avoid a clash with the already loaded `list.scm`.

```
all xs1,xs2,xs3 xs1++(xs2++xs3)eqd xs1++xs2++xs3
```

Then we carry on by

```
(simp "ListAppdAssoc")
(use "InitEqD")
```

6.3.1. *Exercise.* In `list.scm` `ListMap` is introduced by

```
(add-program-constant
 "ListMap" (py "(alpha1=>alpha2)=>list alpha1=>list alpha2"))

(add-infix-display-string "ListMap" "map" 'pair-op)

(add-var-name "phi" (py "alpha1=>alpha2"))

(add-computation-rules
 "phi map(Nil alpha1)" "(Nil alpha2)"
 "phi map y::ys" "phi y::phi map ys")
```

Prove that `map` commutes with `Rv`:

```
(av "f" (py "alpha=>alpha"))
(set-goal "all f,xs (f map Rv xs)eqd Rv(f map xs)")
```

In the proof it is helful to use the theorem `MapAppd`, which can be found in `list.scm`.

6.4. **Defining algebras: binary trees.** We now wish to show how to introduce new algebras; we shall also give one more example on how to use them. The example we shall consider is that of binary trees. First of all we introduce a new algebra, called “bintree” which has constructors “Null” and “Con”. We also add two variables of type “bintree”: “ltree” and “rtree” (for left and right tree).

```
(add-algs "bintree"
          '("bintree" "Null")
          '("bintree=>nat=>bintree=>bintree" "Con"))
(av "ltree" "rtree" (py "bintree"))
```

We then add a new program constant (by using a shortcut, `apc`, for the command `add-program-constant`) and its respective computation rules. `Flatten` takes a tree and produces a list consisting of the labels in the tree, starting with the root label.

```
(apc "Flatten" (py "bintree=>list nat"))
(add-computation-rules
 "Flatten(Null)" "(Nil nat)"
 "Flatten(Con ltree n rtree)"
 "n: ++Flatten ltree++Flatten rtree")
```

To see how this works, one can for example type:

```
(pp (nt (pt "Flatten(Con(Con Null 4 Null)
                       1
                       (Con Null 5(Con Null 7 Null)))")))
```

to obtain `1::4::5::7:` as the list of labels.

6.5. **Inductively defined predicates.** In Minlog we can also introduce inductively generated predicates with the command `add-ids`. An example defining the even numbers inductively is as follows:

```
(add-ids
  (list (list "EvenI" (make-arity (py "nat")) "algEvenI"))
  '("EvenI 0" "InitEvenI")
  '("allnc n(EvenI n -> EvenI(n+2))" "GenEvenI"))
```

The two closure axioms of this inductive definition are `InitEvenI: EvenI 0` and `GenEvenI: allnc n( EvenI n -> EvenI(n + 2))`. In the latter, we have made use of the "non-computational" quantifier `allnc`. We briefly recall that a non-computational quantifier may be used in cases where the variable it quantifies on will not be used (freely) in any term in the proof.[17] Seen purely logically there is no difference between the `all` and `allnc` quantifier. With `algEvenI`, we provide a name for an algebra, corresponding to this inductive definition. It can be omitted, in which case we have an "inductive definition without computational content".

Similarly to the case of simultaneous free algebras, we could use `add-ids` to introduce simultaneously more than one predicate. The case above is a particular example, in which the first occurrence of `list` is followed by only one item.

Let's now see a proof which uses the closure axioms for `EvenI`. This makes essential use of the command `intro`. The command `(intro i . terms)` expects as goal a formula which can be proved using the i-th closure axiom. This axiom is then applied, via `use`, hence `terms` may have to be provided. For example, below by calling `intro 0` we apply the first closure axiom for EvenI, and by calling `intro 1` we apply the second closure axiom.

```
(set-goal "all n EvenI(n+n)")
(ind)
(ng)
(intro 0)
(assume "n" "IH")
```

[17] Consequently, extracted programs (see the next chapter) will not depend on these variables.

```
(ng)
(intro 1)
(use "IH")
```

To conclude this section, we present a proof using induction on the predicate `EvenI`.

```
(set-goal "allnc n(EvenI n -> ex m m+m=n)")
(assume "n" "En")
(elim "En")
```

Here `elim` applies the induction axiom for the inductive definition. For example, if the goal is `all n(EvenI n -> P n)`, we obtain the new goals: `P 0` and `all n((EvenI n & P n) -> P(n+2))`. The commands for the rest of the proof have been explained before.

```
(ex-intro (pt "0"))
(use "Truth")

(assume "n1" "En1" "IH")
(by-assume "IH" "m0" "m0Prop")
(ex-intro (pt "m0+1"))
(simp "<-" "m0Prop")
(use "Truth")
```

6.6. **Totality of program constants.** An important example for an inductively defined predicate is the totality predicate for an algebra, for instance `TotalNat` for the algebra `nat`. It can be created by calling `(add-totality` *`alg-name`*`)`. When the file `nat.scm` is loaded this already has been done and `TotalNat` is inductively defined by the two clauses `TotalNatZero` and `TotalNatSucc`:

$$\texttt{TotalNatZero}\colon \texttt{TotalNat}\,0$$

$$\texttt{TotalNatSucc}\colon \forall^{\mathrm{nc}}_{\hat{n}}(\texttt{TotalNat}\,\hat{n} \to \texttt{TotalNat}(\texttt{Succ}\,\hat{n}))$$

This can be checked by executing

```
(display-idpc "TotalNat")
```

Note that here we have used $\hat{n}$ rather than n as a variable name. At this point it is appropriate to remember that in our intended model (the Scott-Ershov partial continuous functionals) *partial* objects are first class citizens, and hence quantifiers by default range over them. When we want to talk about total objects only, we need to relativize quantifiers to a totality predicate. To make the notation less cumbersome we introduce the convention that if a variable name is followed by a `^`, a general (or partial)) variable is meant. Variable names without a `^` are implicitly restricted to range over total objects only. In fact,

$\forall_x Px$ is just a convenient abbreviation for $\forall^{\text{nc}}_{\hat{x}}(T\hat{x} \to P\hat{x})$. Here T is the totality predicate for the current type.

When adding a program constant and its computation rules the default is that this constant denotes a partial functional. However, often the computation rules are such that it actually is total (i.e., defined for all total arguments). It is good practice to prove totality immediately after defining a program constant.

For example, the program constant `Double` clearly is total. To let Minlog know this fact we have to prove a lemma.

```
;; DoubleTotal
(set-goal (rename-variables
           (term-to-totality-formula (pt "Double"))))
```

The goal then is, as expected,

```
?_1: allnc n^(TotalNat n^ -> TotalNat(Double n^))
```

Now we carry on by assuming the variable and the hypothesis

```
(assume "n^" "Tn")
```

At this point we use the elimination axiom for `TotalNat`

```
(elim "Tn")
(use "TotalNatZero")
(assume "n^1" "Tn1" "IH")
(ng #t)
(use "TotalNatSucc")
(use "TotalNatSucc")
(use "IH")
;; Proof finished.
(save "DoubleTotal")
```

It is important to give this lemma the name `DoubleTotal`, i.e., the name of the program constant followed by `Total`. After the lemma with this name is saved Minlog will know that `Double` is total.

7. Program extraction from proofs

In this section we give some basic examples of program extraction from proofs. We should perhaps mention at this point that program extraction was one of the main original motivations in the development of Minlog. In addition, Minlog features some interesting aspects, as for example it implements a refined version of the so–called A–translation, thus allowing for program extraction from **classical** proofs. An exposition of program extraction from proofs and (modified) A–translation is well beyond the purpose of this tutorial. See for example [2, 1, 11].

7.1. **List reversal.** For our first example of program extraction from proofs, we introduce an inductively defined predicate `RevI` (without computational content) as follows:

```
;; (set! COMMENT-FLAG #f)
;; (libload "nat.scm")
;; (libload "list.scm")
;; (set! COMMENT-FLAG #t)

;; (add-var-name "a" "b" "c" "d" "x" (py "alpha"))
;; (add-var-name "v" "w" "u" "xs" (py "list alpha"))

(add-ids
 (list (list "RevI"(make-arity (py "list alpha")
                               (py "list alpha"))))
 '("RevI(Nil alpha)(Nil alpha)" "InitRev")
 '("all a,v,w(RevI v w  -> RevI(v++a:)(a::w))" "GenRevI"))
```

We first prove that `RevI` satisfies a variant clause[18]:

```
(set-goal "all a,v,w(RevI v w  -> RevI(a::v)(w++a:))")
(assume "a" "v" "w" "Rvw")
(elim "Rvw")
(ng)
;; RevI(a:)(a:)
(use-with "GenRevI" (pt "a") (pt "(Nil alpha)") (pt "(Nil alpha)")
          "InitRevI")
(assume "a1" "v1" "w1" "Rv1w1" "Hyp")
(assert "(a::v1++a1:)eqd(a::v1)++a1:")
 (ng #t)
 (use "InitEqD")
(assume "Assertion1")
(simp "Assertion1")
(assert "(a1::w1)++a:eqd(a1::w1++a:)")
 (ng #t)
 (use "InitEqD")
(assume "Assertion2")
(simp "Assertion2")
(use "GenRevI")
(use "Hyp")
;; Proof finished.
(save "RevIConsAppd")
```

[18] We did not take this variant as the defining clause since our definition will be useful in section 7.3.

Using `RevIConsAppd` we can prove symmetry of `RevI`

```
(set-goal "all v,w(RevI v w -> RevI w v)")
(assume "v" "w" "Rvw")
(elim "Rvw")
(use "InitRevI")
(assume "a" "v1" "w1" "Rv1w1" "Rw1v1")
(use "RevIConsAppd")
(use "Rw1v1")
;; Proof finished.
(save "RevISym")
```

Then we set the goal:

```
(set-goal "all v ex w RevI v w")
```

The proof proceeds by structural induction on lists. We first call `ind`. Subsequently, we tackle the base case by first of all providing a witness, `Nil` of type α, and then by using the first closure axiom for `RevI`.

```
(ind)
(ex-intro (pt "(Nil alpha)"))
(intro 0)
```

The step is proved by first of all calling some standard commands (`assume, by-assume`), then by providing a witness, and finally the second closure axiom is called as usual by an `intro` command.

```
(assume "a" "v" "IH")
(by-assume "IH" "w" "wProp")
(ex-intro (pt "w++a:"))
(use "RevISym")
(intro 1)
(use "RevISym")
(use "wProp")
```

We can finally name the proof we have just completed by writing:

```
(define constr-proof (current-proof))
```

Note that `(current-proof)` stores the latest proof.

We now extract a program from the proof and normalize it as follows:

```
(define eterm (proof-to-extracted-term constr-proof))
(define neterm (rename-variables (nt eterm)))
(pp neterm)
```

We have used `rename-variables` to obtain a more readable term. This “normalized extracted term” `neterm` is the program we are looking for. To display it we write:

```
(pp neterm)
```

The output will be:

```
[xs](Rec list alpha=>list alpha)xs
   (Nil alpha)([x,xs0,xs1]xs1++x:)
```

Here [xs] denotes abstraction on the variable xs, usually also written by use of the λ notation. We observe that the extracted term uses the recursion operator Rec. In more familiar terms, it amounts to a program, which we may call Reverse, defined as follows:

```
Reverse Nil=Nil
Reverse (x :: xs)=(Reverse xs) ++ x:
```

Note that we could have also displayed the program by using the command term-to-scheme-expr, which produces the λ-term corresponding to the program.

To test the program we can "run" it on input [a,b,c,d]:

```
(pp (nt (make-term-in-app-form neterm (pt "a::b::c::d:"))))
```

and obtain the result: d::c::b::a:

7.2. **Program extraction from proofs using inductive definitions with computational content.** In this section we wish to exemplify how to extract a program from a proof by induction of a statement which uses an inductive definition[19]. In fact, an inductive definition with computational content on the proof side corresponds to a free algebra on the program side. Thus we need to provide a name for such an algebra, if a new one has to be generated. However, if there is an already existing algebra with fitting constructors, the name of this algebra can be provided as well. I.e., in our example on the even numbers, which we recall below, we could also take nat. The types of the algebra's constructors corresponds to the clauses of the inductive definition, named InitEven and GenEven.

```
(add-ids
 (list (list "EvenI" (make-arity (py "nat")) "algEvenI"))
 '("EvenI 0" "InitEvenI")
 '("allnc n(EvenI n -> EvenI(n+2))" "GenEvenI"))
```

We recall our proof and extract a program:

```
(set-goal "allnc n(EvenI n -> ex m m+m=n)")
(assume "n" "En")
(elim "En")
```

[19] These inductive definitions are also called "inductive definitions with computational content". Note that there are also inductive definitions without computational content (for example RevI above). For more information on computational content, see for example [11] and [3].

```
(ex-intro (pt "0"))
(use "Truth")
(assume "n1" "En1" "IH")
(by-assume "IH" "m0" "m0Prop")
(ex-intro (pt "m0+1"))
(simp "<-" "m0Prop")
(use "Truth")

(define eterm (proof-to-extracted-term (current-proof)))
(define neterm (rename-variables (nt eterm)))
```

We can see the program by writing:

```
(pp neterm)
```

Here, the extracted program essentially corresponds to the identity function. For comparison we recommend to extract a program using the earlier defined program constants Even/Odd.

```
(set-goal "all n ex m((Even n -> 2*m=n) &
                      (Odd n -> 2*m+1=n))")
```

One first has to prove totality of Even and Odd, then prove their disjointness and finally prove the goal above. A solution to this exercise can be found in Appendix A.1.

As mentioned, we can define the even predicate by an inductive definition without computational content.

```
(add-ids (list (list "EvenNC" (make-arity (py "nat"))))
 '("EvenNC 0" "InitEvenNC")
 '("allnc n(EvenNC n -> EvenNC(n+2))" "GenEvenNC"))
```

Generally, for non-computational (n.c.) inductively defined predicates no algebra name is provided. Important special cases are the "uniform one clause defined" inductive predicates[20] which are non-computational. Examples are Leibniz equality EqD, and uniform variants ExU and AndU of the existential quantifier and conjunction.

One can prove a non-computational version of the lemma above, where we only need to replace the ex-intro command by the general intro, since ExU is inductively defined.

```
(set-goal "allnc n(EvenNC n -> exu m m+m=n)")
(assume "n" "En")
(elim "En")
(intro 0 (pt "0"))
```

[20] An inductive predicates is called uniform one clause defined if it has just one clause with $\forall^{\mathrm{nc}}$ and $\to^{\mathrm{nc}}$ (an n.c. variant of $\to$) only; see the reference manual for details.

```
(use "Truth")
(assume "n1" "En1" "IH")
(by-assume "IH" "m0" "m0Prop")
(intro 0 (pt "m0+1"))
(simp "<-" "m0Prop")
(use "Truth")
```

We conclude this section with a more substantial example of program extraction from proofs involving inductive definitions. Every constructive proof of an existential theorem (or "problem"; cf. [6]) contains – by the very meaning of "constructive proof" – a construction of a solution in terms of the parameters of the problem. To get hold of such a solution we have two methods.

Write-and-verify. Guided by our understanding of how the constructive proof works we directly write down a program to compute the solution, and then formally prove ("verify") that this indeed is the case.

Prove-and-extract. Formalize the constructive proof, and then extract the computational content of this proof in the form of a realizing term t. The soundness theorem guarantees (and even provides a formal proof) that t is a solution to the problem.

In simple cases the two methods are often essentially the same. However, in more complex situations the prove-and-extract method seems to be preferable, for the following reasons.

(i) Dealing with a problem on the proof level makes it possible to use more abstract mathematical tools.
(ii) Generally a better organization of the material becomes possible, which is an essential aspect of a good mathematical analysis of a problem.
(iii) Such a structural approach leads to a better understanding of what is going on, which will make it easier to adapt the proof to a somewhat changed specification.

Consider the problem of recognizing whether a list of left and right parentheses is balanced, and if so produce a generating tree (a.k.a. parse tree). Usually one tackles this problem by the write-and-verify method: one writes such a parser as a shift-reduce syntax analyser, and verifies that it is correct and complete. But we can view it also as a good test for the prove-and-extract method. However, since this example is rather complicated we have relegated its treatment into Appendix A.2.

7.3. **Program extraction from classical proofs.** Finally, we wish to exemplify how to extract programs from classical proofs. Once more, an account of the theory underlying this example exceeds the modest aims of this tutorial, so that we can but refer the inquisitive reader to the literature already mentioned above.

The goal is to prove a classical variant of the statement in Section 7.1. Quite concisely, we set the goal and produce a proof:

```
(set-goal "all v excl w RevI v w")

(assume "v0" "AllNegHyp")
(cut "all u allnc v(v++u eqd v0 ->
                   all w(RevI v w -> bot))")

(assume "claim")
```

Now we can make use of the `claim` to prove the goal:

```
(use "claim"
     (pt "v0") (pt "(Nil alpha)") (pt "(Nil alpha)"))
(ng)
(use "InitEqD")
(intro 0)
```

And prove the `claim` by induction:

```
(ind)
```

The base case is tackled as follows:

```
(assume "v")
(ng)
(assume "v=v0" "w")
(simp "v=v0")
(use "AllNegHyp")
```

As to the step we write:

```
(assume "a" "u" "IH" "v" "EqDHyp" "w" "RHyp")
(use "IH" (pt "v++a:") (pt "a::w"))
(ng)
(use "EqDHyp")
(intro 1)
(use "RHyp")
```

Finally we name the proof, which we conveniently call `class-proof`:

```
(define class-proof (np (current-proof)))
```

Finally we add a variable `g`:

```
(av "g" (py "list alpha=>list alpha"))
```

The reasons for this are purely cosmetic. In fact, g will be the default name in case the extracted program needs a variable of type list alpha=>list alpha. Otherwise the program would use a default variable.

```
(define eterm
  (atr-min-excl-proof-to-structured-extracted-term
   class-proof))
(define neterm (rename-variables (nt eterm)))
```

We display the program and obtain the output:

```
(pp neterm)

[xs]
 (Rec list alpha=>list alpha=>list alpha)xs([xs0]xs0)
 ([x,xs0,g,xs1]g(x::xs1))
 (Nil alpha)
```

Finally:

```
(pp (nt (make-term-in-app-form neterm (pt "a::b::c:"))))
```

This gives the result:

```
c::b::a:
```

To conclude, we would like to remark that this program differs from that obtained in section 7.1. In fact, the program above could be written in a more readable form as follows:

```
Reverse xs0 = reverse-acc xs0 Nil

reverse-acc Nil xs1 = xs1
reverse-acc (x1::xs2) xs4 = reverse-acc xs2 (x1::xs4)
```

The reader can see that this program is linear, hence better than the previous one which is quadratic.

Appendix A. Extraction examples

A.1. **Even and Odd.** We first need to prove totality of Even and Odd

```
;; NatEvenOddTotal
(set-goal "allnc n^(TotalNat n^ -> TotalBoole(Even n^) &
                                  TotalBoole(Odd n^))")
(assume "n^" "Tn")
(elim "Tn")
(split)
(use "TotalBooleTrue")
(use "TotalBooleFalse")
(assume "n^1" "Tn1" "IHn1")
(split)
(ng #t)
(use "IHn1")
(ng #t)
(use "IHn1")
;; Proof finished.
(save "NatEvenOddTotal")

;; EvenTotal
(set-goal
 (rename-variables (term-to-totality-formula (pt "Even"))))
(assume "n^" "Tn")
(use "NatEvenOddTotal")
(use "Tn")
;; Proof finished.
(save "EvenTotal")

;; OddTotal
(set-goal
 (rename-variables (term-to-totality-formula (pt "Odd"))))
(assume "n^" "Tn")
(use "NatEvenOddTotal")
(use "Tn")
;; Proof finished.
(save "OddTotal")
```

Next we prove that Even and Odd are disjoint.

```
;; NatEvenOddDisjunct
(set-goal "all n(Even n -> Odd n -> F)")
(ind)
```

```
;; Base
(ng)
(assume "Useless" "Absurd")
(use "Absurd")
;; Step
(assume "n" "IHn" "E(n+1)" "O(n+1)")
(use-with "IHn" "O(n+1)" "E(n+1)")
;; Proof finished.
(save "NatEvenOddDisjunct")
```

Finally we can prove our goal and extract.

```
(set-goal
 "all n ex m((Even n -> 2*m=n) & (Odd n -> 2*m+1=n))")
(ind)
;; Base
(ex-intro "0")
(split)
(assume "Useless")
(use "Truth")
(assume "Absurd")
(use "Absurd")
;; Step
(assume "n" "IHn")
(by-assume "IHn" "m" "mProp")
(ex-intro "[if (Even n) m (Succ m)]")
(split)
;; Case Odd n
(assume "E(n+1)")
(assert "Even n -> F")
 (assume "En")
 (use "NatEvenOddDisjunct" (pt "n"))
 (use "En")
 (use "E(n+1)")
(assume "Even n -> F")
(simp "Even n -> F")
(ng #t)
(use "mProp")
(use "E(n+1)")
;; Case Even n
(assume "O(n+1)")
(simp "O(n+1)")
(ng #t)
```

```
(use "mProp")
(use "O(n+1)")
;; Proof finished.

(define eterm (proof-to-extracted-term (current-proof)))
(define neterm (rename-variables (nt eterm)))
(pp neterm)
;; [n](Rec nat=>nat)n 0([n0,n1][if (Even n0) n1 (Succ n1)])
```

A.2. **Parsing.** Let E range over expressions formed as lists of left and right parentheses L, R. We are interested in the [13] language of balanced lists of L and R. It is generated by either of the grammars

$$\text{grammar U :} \qquad E ::= \text{Nil} \mid ELER$$
$$\text{grammar S :} \qquad E ::= \text{Nil} \mid LER \mid EE$$

It is not too difficult to see that both grammars generate the same expressions. S appears to be more natural, but its generation trees are not unique: one can always append the empty list Nil. This can be repaired easily by only dealing with non-empty lists. However, a drawback then is that one often wants to specialize general lemmas (like the closure poperty of U below) to the empty list. Therefore we restrict attention to U.

First we formulate the grammar U as an inductively defined predicate over lists x, y, z of parentheses L, R given by the clauses

$$\text{InitU}\colon U(\text{Nil})$$
$$\text{GenU}\colon Ux \to Uy \to U(xLyR)$$

The corresponding free algebra on the program side will be that of binary trees, which we introduce first.

```
(add-algs "bin"
  '("bin" "I")
  '("bin=>bin=>bin" "C"))
```

Since we will work with lists of parentheses, we need the library file `list.scm`.

```
(set! COMMENT-FLAG #f)
(libload "nat.scm")
(libload "list.scm")
(set! COMMENT-FLAG #t)

(add-algs "par" '("L" "par") '("R" "par"))
(add-totality "par")
```

As a preparatory step we first have to prove that ordinary equality = (a boolean valued function) implies Leibniz equality, for the two types `par` and `list par`.

```
;; ParEqToEqD
(set-goal "all par1,par2(par1=par2 -> par1 eqd par2)")
(cases)
(cases)
(assume "Useless")
(use "InitEqD")
(assume "L=R")
(use "EFEqD")
(use "AtomToEqDTrue")
(use "L=R")
(cases)
(assume "R=L")
(use "EFEqD")
(use "AtomToEqDTrue")
(use "R=L")
(assume "Useless")
(use "InitEqD")
;; Proof finished.
(save "ParEqToEqD")

(add-var-name "x" "y" "z" (py "list par"))

;; ListParEqToEqD
(set-goal "all x1,x2(x1=x2 -> x1 eqd x2)")
(ind)
(cases)
(assume "Useless")
(use "InitEqD")
(assume "par1" "x1" "Absurd")
(use "EFEqD")
(use "AtomToEqDTrue")
(use "Absurd")
(assume "par1" "x1" "IH")
(cases)
(assume "Absurd")
(use "EFEqD")
(use "AtomToEqDTrue")
(use "Absurd")
```

```
(assume "par2" "x2" "=Hyp")
(ng "=Hyp")
(assert "x1=x2")
 (use "=Hyp")
(assume "x1=x2")
(assert "par1=par2")
 (use "=Hyp")
(assume "par1=par2")
(drop "=Hyp")
(assert "x1 eqd x2")
 (use "IH")
 (use "x1=x2")
(assume "x1 eqd x2")
(assert "par1 eqd par2")
 (use "ParEqToEqD")
 (use "par1=par2")
(assume "par1 eqd par2")
(elim "x1 eqd x2")
(assume "x^3")
(elim "par1 eqd par2")
(assume "par^3")
(use "InitEqD")
;; Proof finished.
(save "ListParEqToEqD")
```

Now we inductively define a predicate (grammar) U over lists of parentheses.

```
(add-ids
 (list (list "U" (make-arity (py "list par")) "bin"))
 '("U(Nil par)" "InitU")
 '("allnc x,y(U x -> U y -> U(x++L: ++y++R:))" "GenU"))
```

We work with two predicates $\mathrm{RP}(n, x)$ meaning $U(xR^n)$ and $\mathrm{LP}(n, y)$ meaning $U(L^n y)$. For RP we have an inductive definition

$$\mathrm{RP}(0, \mathrm{Nil})$$

$$Uz \to \mathrm{RP}(n, x) \to \mathrm{RP}(n + 1, xzL)$$

We define RP with a parameter predicate to be substituted by U.

```
(add-pvar-name "P" (make-arity (py "list par")))

(add-ids
 (list (list "RP" (make-arity (py "nat") (py "list par"))
        "list"))
 '("RP 0(Nil par)" "InitRP")
 '("allnc n,x,z(P z -> RP n x -> RP(Succ n)(x++z++L:))"
   "GenRP"))
```

The algebra associated with this definition of RP is lists of parentheses.

LP can be defined via a boolean valued function with defining equations

$$\begin{aligned}
\mathrm{LP}(0, \mathrm{Nil}) &= \mathsf{tt} \\
\mathrm{LP}(n+1, \mathrm{Nil}) &= \mathsf{ff} \\
\mathrm{LP}(n, Lx) &= \mathrm{LP}(n+1, x) \\
\mathrm{LP}(0, Rx) &= \mathsf{ff} \\
\mathrm{LP}(n+1, Rx) &= \mathrm{LP}(n, x)
\end{aligned}$$

In Minlog this reads

```
(add-program-constant "LP" (py "nat=>list par=>boole"))

(add-computation-rules
 "LP 0(Nil par)"       "True"
 "LP(Succ n)(Nil par)" "False"
 "LP n(L::x)"          "LP(Succ n)x"
 "LP 0(R::x)"          "False"
 "LP(Succ n)(R::x)"    "LP n x")
```

As mentioned above, it is advisable to prove totality of a program constant immediately after its definition.

```
(set-totality-goal "LP")
(assert
 "allnc x^(TotalList x^ -> allnc n^(TotalNat n^ -> TotalBoole(LP n^ x^)))")
 (assume "x^" "Tx")
 (elim "Tx")
 (assume "n^" "Tn")
 (elim "Tn")
 (use "TotalBooleTrue")
 (assume "n^1" "Useless1" "Useless2")
 (use "TotalBooleFalse")
 (assume "par^" "Tpar")
 (elim "Tpar")
 (assume "x^1" "Tx1" "IHx1" "n^" "Tn")
 (ng #t)
 (use "IHx1")
 (use "TotalNatSucc")
 (use "Tn")
 (assume "x^1" "Tx1" "IHx1" "n^" "Tn")
 (elim "Tn")
 (use "TotalBooleFalse")
 (assume "n^1" "Tn1" "Useless")
 (ng #t)
 (use "IHx1")
 (use "Tn1")
(assume "LPTotalAux" "n^" "Tn" "x^" "Tx")
(use "LPTotalAux")
(use "Tx")
(use "Tn")
;; Proof finished.
(save-totality)
```

Then clearly the following closure property of U holds

$$\mathrm{RP}(n,x) \to U(z) \to \mathrm{LP}(n,y) \to U(xzy).$$

One proves by induction on y that the claim holds for all n.

```
;; ClosureU
(set-goal
 "all y allnc n,x,z((RP (cterm (x^) U x^))n x ->
  U z -> LP n y -> U(x++z++y))")
(ind)
```

In the base case $y = \mathrm{Nil}$ ine uses induction on $\mathrm{RP}(n,x)$.

```
(assume "n" "x" "z" "RP n x")
(elim "RP n x")
;; InitRP
(ng #t)
(auto)
;; GenRP
(ng #t)
(assume "n1" "x1" "z1" "Useless1" "Useless2"
 "Useless3" "Useless4" "Absurd")
(use "Efq")
(use "Absurd")
```

In the step one distinguishes cases on the first character. In case $L :: y$ use the induction hypothesis for $n+1$.

```
(cases)
(ng #t)
(assume "y" "IHy" "n" "x" "z" "RP n x" "U z" "LP(Succ n)y")
(use-with "IHy" (pt "Succ n") (pt "x++z++L:")
 (pt "(Nil par)") "?" "?" "?")
```

```
(use "GenRP")
(use "U z")
(use "RP n x")
(use "InitU")
(use "LP(Succ n)y")
```

In case $R :: y$ again use induction on $\mathrm{RP}(n, x)$. The first RP clause uses Efq, the second one the induction hypothesis on y, `GenU` and equality arguments.

```
(assume "y" "IHy" "n" "x" "z" "RP n x")
(elim "RP n x")

;; First RP clause
(ng #t)
(assume "U z" "Absurd")
(use "Efq")
(use "Absurd")

;; Second RP clause.  Uses IHy, GenU and equality arguments.
(assume "n1" "x1" "z1" "U z1" "RP n1 x1" "IH" "U z")
(ng #t)
(simp (pf "x1++z1++(L::z)++(R::y)=x1++z1++(L::z)++R: ++y"))
(simp (pf "x1++z1++(L::z)=x1++(z1++(L::z))"))
(simp (pf "x1++(z1++(L::z))++R: =x1++(z1++(L::z)++R:)"))
(use "IHy")
(use "RP n1 x1")
(use-with "GenU" (pt "z1") (pt "z") "U z1" "U z")
(simp "ListAppdAssoc")
(simp "ListAppdAssoc")
(simp "ListAppdAssoc")
(use "Truth")
(simp "ListAppdAssoc")
(use "Truth")
(ng #t)
(use "Truth")
;; Proof finished
(save "ClosureU")
```

In particular we have $\mathrm{LP}(0, y) \to U(y)$.

Conversely one can easily prove $U(y) \to \mathrm{LP}(0, y)$ by induction on U. One needs a property of `LP` first

```
;; LPProp
(set-goal "all x,y,n,m(LP n x -> LP m y -> LP(n+m)(x++y))")
```

```
(ind)
;; 2,3
(ind)
;; 4,5
(cases)
(cases)
(auto)
;; 5
(ng)
(cases)
(assume "y" "IHy")
(ng)
(assume "n" "m" "Hyp1" "Hyp2")
(use-with "IHy" (pt "n") (pt "Succ m") "Hyp1" "Hyp2")
(assume "y" "IHy" "n")
(cases)
(assume "Hyp1" "Absurd")
(use "Efq")
(use "Absurd")
(ng)
(use "IHy")
;; 3
(cases)
(assume "x" "IHx")
(ng)
(assume "y" "n" "m" "Hyp1" "Hyp2")
(use-with "IHx" (pt "y") (pt "Succ n") (pt "m") "Hyp1" "Hyp2")
(assume "x" "IHx" "y")
(cases)
(assume "m" "Absurd" "Hyp1")
(use "Efq")
(use "Absurd")
(use "IHx")
;; Proof finished.
(save "LPProp")
```

Using `LPProp` one can prove

```
;; Soundness
(set-goal "all y(U y -> LP 0 y)")
(assume "z" "IdHyp")
(elim "IdHyp")
(use "Truth")
```

```
(assume "x" "y" "Ux" "LP 0 x" "Uy" "LP 0 y")
(simp "<-" "ListAppdAssoc")
(use-with "LPProp" (pt "x") (pt "L::y++R:") (pt "0") (pt "0")
  "LP 0 x" "?")
(ng #t)
(use-with "LPProp" (pt "y") (pt "R:") (pt "0") (pt "1")
  "LP 0 y" "Truth")
;; Proof finished.
(save "Soundness")
```

From `ClosureU` we obtain

```
;; Completeness
(set-goal "all y(LP 0 y -> U y)")
(assume "y" "LP 0 y")
(use-with "ClosureU" (pt "y") (pt "0")
           (pt "(Nil par)")  (pt "(Nil par)")
         "?" "InitU" "LP 0 y")
(use "InitRP")
;; Proof finished.
(save "Completeness")
```

Hence the test $\mathrm{LP}(0, y)$ is correct (all y in U satisfies it) and complete (it implies y in U). Because of $\mathrm{LP}(0, y) \leftrightarrow U(y)$ we have a decision procedure for U. With p a boolean variable we can express this by a proof of

$$\forall_y \exists_p ((p \to U(y)) \land ((p \to \mathbf{F}) \to U(y) \to \mathbf{F}))$$

```
(add-var-name "p" (py "boole"))

;; Parse
(set-goal "all y ex p((p -> U y) & ((p -> F) -> U y -> F))")
(assume "y")
(ex-intro "LP 0 y")
(split)
(use "Completeness")
(assume "LP 0 y -> F" "Uy")
(use "LP 0 y -> F")
(use "Soundness")
(use "Uy")
;; Proof finished.
(save "Parse")
```

The computational content of this proof is a parser for U. Given y it returns a boolean saying whether or not y is in U, and if so it also returns a generation tree (a.k.a. parse tree) for $U(y)$.

To extract the computational content we need to "animate" the theorems `ClosureU` and `Completeness`, or more precisely the automatically generated program constants `cClosureU` and `cCompleteness` abbreviating their computational content. These constants will be unfolded under normalization once the theorems are animated.

```
(animate "ClosureU")
(animate "Completeness")

(add-var-name "a" (py "bin"))
(add-var-name "as" (py "list bin"))
(add-var-name "f" (py "list bin=>bin=>bin"))

(define eterm
 (proof-to-extracted-term (theorem-name-to-proof "Parse")))
(define neterm-Parse (rename-variables (nt eterm)))
(ppc neterm-Parse)
```

Here is the term extracted from the proof above.

```
[x]LP 0 x@
 (Rec list par=>list bin=>bin=>bin)x
 ([as,a][case as (Nil -> a) (a0::as0 -> I)])
 ([par,x0,f,as,a]
   [case par
     (L -> f(a::as)I)
     (R -> [case as (Nil -> I) (a0::as0 -> f as0(C a0 a))])])
 Nil
 I
```

Since this term involves the recursion operator it is not easy to read. To grasp its meaning we rewrite it. It amounts to applying a function g to x, Nil and I, where

$$\begin{aligned}
g(\mathrm{Nil}, as, a) \quad &= \begin{cases} a & \text{if } as = \mathrm{Nil} \\ I & \text{else} \end{cases} \\
g(L :: x, as, a) &= g(x, a :: as, I) \\
g(R :: x, as, a) &= \begin{cases} I & \text{if } as = \mathrm{Nil} \\ g(x, as_0, C(a_0, a)) & \text{if } as = a_0 :: as_0 \end{cases}
\end{aligned}$$

In $g(x, as, a)$ the first argument x is a list of parentheses L, R to be parsed. The second argument as is a stack of parse trees, and the third a is the working memory of the parser which stores the parse tree being generated. Initially g is called with x, the empty stack Nil and the empty parse tree I.

Recall the grammar U. We read x from left to right. When an L occurs, the current parse tree a (corresponding to E_0 in E_0LE_1R) is pushed onto the stack, and then g starts generating a parse tree for E_1, with the empty parse tree I in its working memory. Now suppose R occurs in x. If the stack is Nil, return the empty parse tree I. If not, pop the top element a_0 from the stack. Then g starts generating a parse tree from the rest of x, the tail as_0 of the stack, and as current parse tree $C(a_0, a)$ in its working memory.

If the input x is empty, in case the stack as is empty as well the current parse tree a is returned, and otherwise the empty parse tree I.

To test our extracted `neterm-Parse` we use some simple Scheme functions. `(generate-seq n)` generates a list of 2^n infinite sequences starting with all possible variations of n digits and continuing with 0.

```
(define (generate-seq n)
  (if (= n 0)
      (list (lambda (n) 0))
      (foldr (lambda (f l)
                 (cons (lambda (n) (if (= n 0) 0 (f (- n 1))))
                       (cons (lambda (n) (if (= n 0) 1 (f (- n 1))))
                             l)))
             '()
             (generate-seq (- n 1)))))
```

`(first f n)` returns a list of (f 0),(f 1),...,(f n-1).

```
(define (first f n)
  (if (= n 0)
      '()
       (cons (f 0)
             (first (lambda (n) (f (+ n 1))) (- n 1)))))
```

We also use

```
(define (blist-to-lpar-term blist)
  (if (null? blist)
      (pt "(Nil par)")
      (mk-term-in-app-form
       (pt "(Cons par)")
       (if (zero? (car blist)) (pt "L") (pt "R"))
       (blist-to-lpar-term (cdr blist)))))

(define (generate-lpar-terms n)
  (let* ((seq (generate-seq n))
         (01lists (map (lambda (f) (first f n)) seq))
         (reduced-01lists
          (list-transform-positive 01lists
            (lambda (l)
              (and (zero? (car l))
                   (not (zero? (car (last-pair l)))))))))
    (map blist-to-lpar-term reduced-01lists)))
```

Now we can test `neterm-Parse` on all lpar-terms of length l.

```
(define (test-parser-term parser-term . l)
  (let ((len (if (null? l) 4 (car l))))
    (map (lambda (lpar-term)
           (display "Testing on ")
           (display (term-to-string lpar-term))
           (let* ((pairterm (nt (make-term-in-app-form
                                 parser-term lpar-term)))
                  (lterm
                   (term-in-pair-form-to-left pairterm))
                  (rterm
                   (term-in-pair-form-to-right pairterm)))
             (if (and (term-in-const-form? lterm)
                      (string=?
                       "True"
                       (const-to-name
                        (term-in-const-form-to-const
                         lterm))))
                 (begin (display " Parse tree: ")
                        (display (term-to-string rterm)))
                 (display " No"))
             (newline)))
         (generate-lpar-terms len)))
    *the-non-printing-object*)
```

We obtain for $l = 6$

```
(test-parser-term neterm-Parse 6)
```

The result is

```
Testing on L::R::R::R::R::R: No
Testing on L::L::R::R::R::R: No
Testing on L::R::L::R::R::R: No
Testing on L::L::L::R::R::R: Parse tree: C I(C I(C I I))
Testing on L::R::R::L::R::R: No
Testing on L::L::R::L::R::R: Parse tree: C I(C(C I I)I)
Testing on L::R::L::L::R::R: Parse tree: C(C I I)(C I I)
Testing on L::L::L::L::R::R: No
Testing on L::R::R::R::L::R: No
Testing on L::L::R::R::L::R: Parse tree: C(C I(C I I))I
Testing on L::R::L::R::L::R: Parse tree: C(C(C I I)I)I
Testing on L::L::L::R::L::R: No
Testing on L::R::R::L::L::R: No
Testing on L::L::R::L::L::R: No
Testing on L::R::L::L::L::R: No
Testing on L::L::L::L::L::R: No
```

Appendix B. Useful Commands

B.1. Emacs.

- Start Emacs: `emacs &`
- Leave Emacs: C-x C-c
- Split a window in two: `C-x 2`
- Move to another Buffer: `C-x b` (then specify the Buffer's name)
- Move to another window: `C-x o`
- Load a file: `C-x C-f` (then give a name of a file with extension `.scm`)
- Save a file: `C-x C-s`
- Exit from the Minibuffer: `C-g`

B.2. Scheme.

- Load (Petite) Scheme: `M-x run-petite`
- Evaluate a Scheme expression: `C-x C-e`
- Evaluate a region: mark the region and then `C-c C-r`
- Kill a process: `C-c C-c`
- Leave the Debug: `r`
- End a Scheme session: `(exit)`
- Comment: `;`

`C = Control` (or `Strg`), `M = Meta` (or `Edit` or `Esc` or `Alt`).

B.3. **Minlog.** The following is a list of commands which could be used in a "standard" interactive proof with Minlog. Rather than explaining the commands in detail (many of them have been demonstrated in the above tutorial), we shall write them down, often with a short description of their use gathered from the reference manual. The reader is advised to check the full details with the reference manual.

B.3.1. *Some declarations needed to start a proof.*

```
(add-tvar-name name1 ...)
(add-algs ...)
(add-var-name name1 ... type)
(add-predconst-name name1 ... arity)
(add-pvar-name name1 ... type)
(add-program-constant name type <rest>)
(add-computation-rule lhs rhs)
(add-rewrite-rule lhs rhs)
(add-global-assumption name formula)   (abbr. aga)
```

For each introduction command above there corresponds another one having the effect of removing the item so introduced (constants, variables, etc). For example:

```
(remove-predconst-name name1 ...)
```

There are also numerous display commands, in particular the following:

```
(display-pconst name1 ...).
(display-alg alg-name1 ...)
(display-idpc idpc-name1 ...)
(display-global-assumptions string1 ...)
(display-theorems string1 ...)
```

For types, terms and formulas there is a command `(pp object)` (for pretty-print), which tries to insert useful line breaks. Variants are `(ppc object)` (for pretty-print with case display) and `(pp-subst substitution)` (for pretty-printing substitutions).

`rename-variables` renames bound variables in terms, formulas and comprehension terms.

B.3.2. *Goals.*

(1) `(set-goal formula)` where *formula* needs to be closed (if it not so, then universal quantifiers will be inserted automatically).
(2) `(normalize-goal . ng-info)` (abbr. `ng`) takes optional arguments `ng-info`. If there are none, the goal formula and all hypotheses are normalized. Otherwise exactly those among the hypotheses and the goal formula are normalized whose numbers (or names, or just `#t` for the goal formula) are listed as additional arguments.
(3) `(display-current-goal)` (abbr. `dcg`).

B.3.3. *Generating interactive proofs.* Implication

`(assume x1...)`

moves the antecedent of a goal in implication form to the hypotheses. The hypotheses, *x1*..., should be identified by numbers or strings.

`(use x)`

where x is

- a number or string identifying a hypothesis from the context,
- the string "Truth",
- the name of a theorem or global assumption.
- a closed proof,
- a formula with free variables from the context, generating a new goal.

Conjunction

`(split)`

expects a conjunction $A \wedge B$ as goal and splits it into two new goals, A and B.

`(use x .  elab-path)`

where x is as in the description of the `use` command for implication and *elab-path* consists of `'left` or `'right`.

Universal Quantifier

`(assume x1...)`

moves universally quantified variables into the context. The variables need to be named (by using previously declared names of the appropriate types).

`(use x .  terms)`

where x is as in the case of implication and the optional *terms* is here a list of terms. When pattern unification succeeds in finding appropriate instances for the quantifiers in the goal, then these instances will be automatically inserted. However, one needs to explicitly provide terms for those variables that cannot be automatically instantiated by pattern unification.

Existential Quantifier

`(ex-intro term)`

by this command the user provides a term to be used for the present (existential) goal.

`(ex-elim x)`,

where x is

- a number or string identifying an existential hypothesis from the context,
- the name of an existential global assumption or theorem,
- a closed proof on an existential formula,

- an existential formula with free variables from the context, generating a new goal.

Classical Existential Quantifier

`(exc-intro` *`terms`*`)`

this command is analogous to `(ex-intro)`, but it is used in the case of a classical existential goal.

`(exc-elim` *`x`*`)`

this corresponds to `(ex-elim)` and applies to a classical existential quantifier.

B.3.4. *Other general commands.* `(use-with` *`x`* `.` *`x-list`*`)` is a more verbose form of `use`, where the terms are not inferred via unification, but have to be given explicitly. Here x is as in `use`, and *x-list* is a list consisting of

- a number or string identifying a hypothesis form the context,
- the name of a theorem or global assumption,
- a closed proof,
- the string "?" generating a new goal,
- `'left` or `'right`,
- a term, whose free variables are added to the context.

`(inst-with` *`x`* `.` *`x-list`*`)`

does for forward chaining the same as `use-with` for backward chaining. It adds a new hypothesis which is an instance of a selected hypothesis or of a theorem. Here x and *x-list* are as in `use-with`.

`(inst-with-to` *`x`* `.` *`x-list`* `name-hyp)`

expects a string as its last argument, to name the newly introduced instantiated hypothesis.

`(cut` *`A`*`)`

replaces the goal B by the two new goals A and $A \to B$, with $A \to B$ to be proved first. Note that the same effect can also be produced by means of the `use` command.

`(assert` *`A`*`)`

replaces the goal B by the two new goals A and $A \to B$, with A to be proved first.

`(ind)`

expects a goal $\forall_{x^\rho} A$ with ρ an algebra. If $c_1, \dots, c_n$ are the constructors of the algebra ρ, then `(ind)` will generate n new goals:

$$\forall_{\vec{x}_i}(A[x := x_{1i}] \to \cdots \to A[x := x_{ki}] \to A[x := c_i\vec{x}_i]).$$

`(simind` *`all-formula1`*`...)`

expects a goal $\forall_{x^\rho} A$ with ρ an algebra. The user provides other formulas to be proved simultaneously with the given one.

`(cases)`
expects a goal $\forall_{x^\rho} A$ with ρ an algebra. Assume that $c_1, \dots, c_n$ are the constructors of the algebra ρ. Then n new (simplified) goals $\forall_{\vec{x}_i} A[x := c_i\vec{x}_i]$ are generated.

`(simp x)`
expects a known fact of the form $r^{\mathbf{B}}$, $\neg r^{\mathbf{B}}$, $t = s$ or $t \approx s$. In case $r^{\mathbf{B}}$, the boolean term r in the goal is replaced by T, and in case $\neg r^{\mathbf{B}}$ it is replaced by F. If $t = s$ (resp. $t \approx s$), the goal is written in the form $A[x := t]$. Using Compat-Rev (i.e. $\forall_{x,y}(x = y \to Py \to Px)$) (resp. Eq-Compat-Rev (i.e. $\forall_{x,y}(x \approx y \to Py \to Px)$)) the goal $A[x := t]$ is replaced by $A[x := s]$, where P is $\{ x \mid A \}$, x is t and y is s. Here `x` is

- a number or string identifying a hypothesis form the context,
- the name of a theorem or global assumption, or
- a closed proof,
- a formula with free variables from the context, generating a new goal.

`(name-hyp` *`i x1`*`)`
expects an index i and a string. Then a new goal is created, which differs from the previous one only in display aspects: the string names the ith hypothesis.

`(drop .  x-list)`,
hides (but does not erase) the hypothesis listed in `x-list`. If `x-list` is empty, all hypotheses are hidden.

`(by-assume` *`x yu`*`)`
is used when proving a goal G from an existential hypothesis $ExHyp\colon \exists y A$. It corresponds to saying "by $ExHyp$ assume we have a y satisfying A". Here `x` identifies an existential hypothesis, and we assume the variable y and the kernel A (with label u). This command corresponds to the sequence `(ex-elim x)`, `(assume` *`y u`*`)`, `(drop x)`.

`(intro i . terms)`
expects as goal an inductively defined predicate. The i-th introduction axiom for this predicate is applied, via `use` (hence `terms` may have to be provided).

`(elim` *`idhyp`*`)`
Recall that $I\vec{r}$ provides (i) a type substitution, (ii) a predicate instantiation, and (iii) the list $\vec{r}$ of argument terms. In `(elim` *`idhyp`*`)` *idhyp* is, with an inductively defined predicate I,

- a number or string identifying a hypothesis $I\vec{r}$ form the context
- the name of a global assumption or theorem $I\vec{r}$;
- a closed proof of a formula $I\vec{r}$;

- a formula $I\vec{r}$ with free variables from the context, generating a new goal.

Then the (strengthened) elimination axiom is used with $\vec{r}$ for $\vec{x}$ and *idhyp* for $I\vec{r}$ to prove the goal $A(\vec{r})$, leaving the instantiated (with $\{\, \vec{x} \mid A(\vec{x}\,)\,\}$) clauses as new goals.

`(elim)`

expects a goal $I\vec{r} \to A(\vec{r})$. Then the (strengthened) clauses are generated as new goals, via `use-with`.

`(undo)` or `(undo` *n*`)`

has the effect of cancelling the last step in a proof, or the last n steps, respectively.

B.3.5. *Automation and search.*

`(strip)`

moves all universally quantified variables and hypotheses of the current goal into the context.

`(strip n)`

does the same as `(strip)` but only for n variables or hypotheses.

`(proceed)`

automatically refines the goal as far as possible as long as there is a unique proof. When the proof is not unique, it prompts us with the new refined goal, and allows us to proceed in an interactive way.

`(prop)`

searches for a proof of the stated goal. It is devised for propositional logic only.

`(search` *m (name1 m1)* `...)`

expects for m a default value of multiplicity (i.e. a positive integer stating how often the assumptions are to be used). Here *name1* ... are

- numbers or names of hypotheses from the present context or
- names of theorems or global assumptions,

and *m1* ... indicate the multiplicities of the specific *name1* .... To exclude a hypothesis one can list it with multiplicity 0.

`(auto` *m (name1 m1)* `...)`

It can be convenient to automate (the easy cases of an) interactive proof development by iterating `search` as long as it is successful in finding a proof. Then the first goal where it failed is presented as the new goal. `auto` takes the same arguments as `search`.

B.3.6. *Displaying proofs objects.* There are many ways to display a proof. We normally use `display-proof` for a linear representation, showing the formulas and the rules used. We also provide a (hopefully) readable type-free lambda expression via `proof-to-expr`, and we can add useful information with either `proof-to-expr-with-formulas` or `proof-to-expr-with-aconsts`. In case the optional proof argument is not present, the current proof is taken instead.

`(display-proof .` *`opt-proof`*`)` abbreviated `dp`,
`(display-normalized-proof .` *`opt-proof`*`)` abbreviated `dnp`,
`(proof-to-expr .` *`opt-proof`*`)`,
`(proof-to-expr-with-formulas .` *`opt-proof`*`)`,
`(proof-to-expr-with-aconsts .` *`opt-proof`*`)`.

Here `display-normalized-proof` normalizes the proof first. When in addition one wants to check the correctness of the proof, use

`(check-and-display-proof .` *`opt-proof-and-ignore-deco-flag`*`)`

abbreviated `cdp`. `ignore-deco-flag` is set to true as soon as the present proof argument proves a formula of nulltype.

B.3.7. *Searching for theorems.* It is a practical problem to find existing theorems or global assumptions relevant for the situation at hand. To help searching for those we provide

`(search-about` *`symbol-or-string`* `.` *`opt-strings`*`)`.

It searches in `THEOREMS` and `GLOBAL-ASSUMPTIONS` for all items whose name contains each of the strings given, excluding the strings Total Partial CompRule RewRule Sound. It one wants to list all these as well, take the symbol 'all as first argument.

References

[1] Holger Benl, Ulrich Berger, Helmut Schwichtenberg, Monika Seisenberger, and Wolfgang Zuber, *Proof theory at work: Program development in the Minlog system*, Automated Deduction – A Basis for Applications (W. Bibel and P.H. Schmitt, eds.), Applied Logic Series, vol. II: Systems and Implementation Techniques, Kluwer Academic Publishers, Dordrecht, 1998, pp. 41–71. 7

[2] Ulrich Berger, Wilfried Buchholz, and Helmut Schwichtenberg, *Refined program extraction from classical proofs*, Annals of Pure and Applied Logic **114** (2002), 3–25. 7

[3] Ulrich Berger and Monika Seisenberger, *Program extraction via typed realisability for induction and coinduction*, Ways of Proof Theory: Festschrift for W. Pohlers (R. Schindler, ed.), Ontos Verlag, 2010. 19

[4] Roy Dyckhoff, *Contraction–free sequent calculi for intuitionistic logic*, The Journal of Symbolic Logic **57** (1992), 793–807. 5

[5] Jörg Hudelmaier, *Bounds for cut elimination in intuitionistic propositional logic*, Ph.D. thesis, Mathematische Fakultät, Eberhard–Karls–Universität Tübingen, 1989. 5

[6] Andrey N. Kolmogorov, *Zur Deutung der intuitionistischen Logik*, Math. Zeitschr. **35** (1932), 58–65. 7.2

[7] Dale Miller, *A logic programming language with lambda–abstraction, function variables and simple unification*, Journal of Logic and Computation **2** (1991), no. 4, 497–536. 5.2

[8] Helmut Schwichtenberg, *Proof search in minimal logic*, Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004, Linz, Austria, September 2004, Proceedings (B. Buchberger and J.A. Campbell, eds.), LNAI, vol. 3249, Springer Verlag, Berlin, Heidelberg, New York, 2004, pp. 15–25. 5.2

[9] ———, *A theory of computable functionals*, 2010, `http://www.minlog-system.de`. 2, 8

[10] ———, *Minlog reference manual*, 2012, `http://www.minlog-system.de`. 2, 3.1.2, 3.1.4, 6.1.1, 11, 6.3

[11] Helmut Schwichtenberg and Stanley S. Wainer, *Proofs and computations*, Perspectives in Logic, Association for Symbolic Logic and Cambridge University Press, 2012. 2, 13, 6.3, 7, 19

[12] Viggo Stoltenberg-Hansen, Edward Griffor, and Ingrid Lindström, *Mathematical theory of domains*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1994. 13

[13] Walther von Dyck, *Gruppentheoretische Studien*, Mathematische Annalen **20** (1882), 1–44. A.2

Index