A TUTORIAL FOR MINLOG, VERSION 4.0

L. CROSILLA

1. Introduction

This is a tutorial for the interactive proof system MINLOG, version 4.0, developed by Helmut Schwichtenberg and members of the logic group at the University of Munich (http://www.mathematik.uni-muenchen.de/∼logik/welcome.html).

MINLOG is implemented in Scheme and runs under every Scheme version supporting the Revised[5] Report on the Algorithmic Language 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 very 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 [**?**]. The papers listed in the MINLOG web page also provide a more detailed and advanced description of the system. In addition, the MINLOG distribution comes equipped with a directory of examples, to which the user is referred. The source code finally provides the ultimate reference.

In order to use MINLOG, one essentially needs a shell in which to run MINLOG and also an editor in which to edit and keep a record of the commands for later sessions. In this tutorial we shall refer to GNU Emacs. While working with Emacs, the ideal would be to split the window in two parts, one with the file in which to store the commands, and the other with the MINLOG interactive section taking place. For this it is recommended to use the startup script *˜/minlog/minlog* which takes files as (optional) arguments. For example

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

opens a new Emacs-Window which is split into two parts. The upper part containes the file *(Buffer file.scm)* whereas the lower part shows the Minlogresponse *(Buffer *minlog*)*.

I would like to thank Professor Schwichtenberg for suggesting to write this tutorial and for useful comments as well as for providing the example on search. Monika Seisenberger commented on an early version of the tutorial proposing improvements and provided some more examples. In writing this tutorial I took inspiration from Martin Ruckert's Tutorial for an earlier version of MINLOG..

If you have already an open emacs window and do not want to open a new one then you can invoke minlog by loading the file *minlog.el*:

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

REMARKS:

In both cases the file *init.scm* is loaded.

In the description above I assumed tacitly that you are using an UNIX-like operating system and that minlog is installed in ~/minlog, where ~ denotes as usual your home directory.

To execute a command of your file, one simply places the cursor at the end of it (after the closed parenthesis), and types `Ctrl-x Ctrl-e`. In general, `Ctrl-x Ctrl-e` will enable us to process any command we type in `tutorial.scm`, but one at the time. To process a whole series of commands, one can highlight the region of interest and type `Ctrl-c Ctrl-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, type `(exit)` to end a SCHEME session and `Ctrl-x Ctrl-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 of association to the right of the parentheses, which is assumed by MINLOG. Therefore the formula above becomes: $(A \to B \to C) \to (A \to B) \to A \to C$. It is very important, especially in the beginning, to pay the maximum attention to this and similar conventions to prevent mistakes. It is probably a good idea to rather exceed in parenthesis in the first examples. MINLOG will automatically delete the parenthesis which are not needed, therefore facilitating the reading.

Making a sketch of the proof. We need in the first place to make an informal plan on how to prove this tautology. While making this 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, until 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. 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. The proof is completed when the list of goals is empty.

In this case the tautology we want to prove is a series of implications, hence we will have to make 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 $(A \to B) \to A \to C$ is an implication and can be treated in the same way; so we now assume $A \to B \to C$ and $A \to B$ and want 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 under the assumption $A \to B \to C$, we simply need to prove both A and B under the same assumption. Obviously A is proved, as it is one of our assumptions, and B immediately follows from $A \to B$ and A.

Writing the formula. Once we have an idea on how to prove the formula, we can start implementing the proof in MINLOG. The initial step would then be to write the formula in MINLOG. For this purpose, we declare three predicate constants A, B and C by writing:

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

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

```
; ok, predicate constant A: (arity) added
; ok, predicate constant B: (arity) added
; ok, predicate constant C: (arity) added
>
```

Subsequently, we write the formula and give it a name, say `distr` (for distributivity of implication):

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

This command has the effect of defining a new variable `distr` and attaching to it the SCHEME term which is produced by the function `pf` applied to the formula we entered. In fact, the function `pf`, short for "parse formula", takes a string as argument and returns a SCHEME term. This SCHEME term is the way MINLOG stores the string, or, equivalently, it 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.

Implementing the Proof. We now want to prove this formula with MINLOG. In order to do this we clearly need to set the formula as our goal.

Setting the goal. Typically, the goals in a proof will be numbered and the top goal will be denoted by the number 1, preceded by a question mark.

To set `distr` as our goal, we type:

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

MINLOG will print:

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

The proof. We have seen in the sketch of the proof that the first step for proving the tautology is 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 needed in order to identify and name the hypothesis. MINLOG will 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 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 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 `(display-pterm)` and `(display-proof-expr)`. See the manual for a description of the various `display` commands available in MINLOG.

We observe that an alternative to using the `define` command at the beginning of the proof and then separately setting the goal, would be to directly set the formula one wants to prove as a goal, that is writing `(set-goal (pf "(A -> B -> C) -> (A -> B) -> A -> C"))`. Note also that the first three `assume` commands could be replaced by only one, i.e.: `(assume 1 2 3)`. 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 to the extent that in case of rather complex proofs, it is convenient to name specific hypothesis, in place of making use of bare numbers. One than can simply use the `assume` commands, 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 [**?**] 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 now presented.

3.2. **A second example: classical logic.** MINLOG implements minimal logic. If we want to prove a proposition which is true in classical logic but not in minimal logic, we explicitly need to state and use principles which are classical in nature. In the following example we shall use Stability, which is added to MINLOG as a global assumption. Roughly speaking, a global assumption is a proposition which can be recalled at any time, if needed, and whose proof does not concern us at the moment (hence it can also be an assumption with no proof). In order to check which global assumptions we have at our disposal we type: `(display-global-assumptions)`. To check a particular global assumption whose name we already know, we write the above command followed by the name of the assumption we want to check, e.g.:

`(display-global-assumptions "Stab-Log")`. Of course we can also introduce our own global assumptions and remove them at any time (see the reference manual for the specific commands).

Stability is the logical law for which, for any proposition A, $\neg\neg A \to A$ holds. In MINLOG, $\neg A$ is defined to be $A \to \bot$, i.e. A implies falsum. Stability is therefore the following proposition: $((A \to \bot) \to \bot) \to A$.

Suppose we want to prove the tautology:

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

which is known as *Peirce formula*. For this second 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 example, 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, because in order to prove A, we will assume that its negation holds and try to get 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 to another global assumption, the principle of "ex falsum quodlibet". This principle allows one to conclude any formula from a proof of falsum, i.e. it is the principle: $\bot \to A$, for arbitrary A.

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

```
(add-predconst-name "A" "B" (make-arity))
(define peirce-formula  (pf "((A -> B) -> A) -> A"))
(set-goal peirce-formula)
(assume 1)
```

We obtain:

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

We now apply Stability, which is stored in MINLOG with the name `Stab-Log`, 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 "Stab-Log")

; 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 falsum quodlibet: if we want to prove B, we can instead prove falsum, since from falsum anything follows, and in particular B. Therefore our goal can be updated to $\bot$ by the following instance of use:

```
(use "Efq-Log")

; 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.
>
```

3.3. **Conjunction.** To conclude this section on propositional logic, we give a short example of a tautology which uses conjunction. We want to prove

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

We shall simply record the code of our MINLOG proof, asking the reader to check MINLOG's reply at each step. A few comments will be added at the end.

```
(add-predconst-name "A" "B" (make-arity))
(set-goal (pf "(A & B) -> (B & A)"))
(assume 1)
(split)
(use 1)
(use 1)
```

The command `(split)` operates on the goal if it is a conjunction and it has the effect of splitting it into its two components. The command `use` is utilized to obtain the left (respectively the right) conjunct in the assumption and "`use`" it to derive the goal.

The reader is encouraged to try and prove other examples of tautologies.

4. Predicate logic

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

Suppose we want to prove that every total relation which is symmetric and transitive is reflexive. For simplicity we shall work with natural numbers. We hence want to prove the following statement:

$$\begin{aligned}&\forall n\,\forall m\,(Rnm \to Rmn) \wedge \forall n\,\forall m\,\forall k\,(Rnm \wedge Rmk \to Rnk)\\ &\to \forall n\,(\exists m\,Rnm \to Rnn),\end{aligned}$$

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

Before starting to prove the claim, we observe that we can equivalently espress it by another formula which is simpler to prove in MINLOG, e.g. by one in which

the conjunctions have been replaced by appropriate implications. That is, we can instead prove the following formula:

$$(\forall n,\, m.Rnm \to Rmn) \to (\forall n,\, m,\, k.Rnm \to Rmk \to Rnk)$$
$$\to \forall n,\, m.Rnm \to Rnn,$$

where the "." is used to indicate the scope of a quantifier, with the convention that it binds as far as possible.

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. However, there might be cases in which one prefers to prove a more complex formula, for example when proving a lemma which is then used in the proof of a more intricate theorem. For completeness and for a comparison, we shall also record a proof of the original goal, at the end of this section.

Since the predicate R is required to vary on natural numbers, we first of all load a file, already available with the distribution, which introduces the algebra of natural numbers and some operations on them, like for example addition. This is obtained by typing:

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

We can now introduce the constant R. We also want to facilitate our work a bit further and separately introduce the two assumptions $\forall n\, m.Rnm \to Rmn$ and $\forall n\, m\, k.Rnm \to Rmk \to Rnk$. We do this by means of a `define` command, which enables us to give a name to each assumption. We then use these name to make a formula in implication form.

In the following `py` is the analogous for types of the function parse formula. Note also that the file `nat.scm` already introduces m, n and k as "default" variables on the natural numbers, hence we do not need to explicitly declare them here, too.

```
(add-predconst-name "R" (make-arity (py "nat") (py "nat")))
(define Symm (pf "all n,m.R n m  -> R m n"))
(define Tran (pf "all n,m,k.R n m -> R m k -> R n k"))
```

We now state the goal:

```
(set-goal (mk-imp Symm Tran (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 in this specific 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, this strategy can be quite useful.

We now observe that the goal is an implication, so that the first step would be to write `(assume "Symm" "Tran")`. By this command we would obtain a universally quantified formula and we would then need to proceed to eliminate the quantifiers. This can be accomplished by another `assume` command in which we specify two natural numbers. For simplicity we here fix the natural numbers n and m. So we

would write `(assume "n" "m")`. This would produce an implication which would also need to be eliminate by another `assume` command, say `(assume 1)`. We can put all these commands together by writing:

```
(assume "Symm" "Tran" "n" "m" 1)

; ok, we now have the new goal
; ?_2: R n n from
;   Symm:all n,m.R n m -> R m n
;   Tran: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 `Tran`, then the goal can be obtained by an instance of `Symm`, and the proof is easily completed. We here utilize `use` by additionally providing a term, `"m"`, which will instantiate the only variable which can not be automatically inferred by unification. In the following `pt` stands for parse term.

```
(use "Tran" (pt "m"))

; ?_4: R m n from
;   Symm:all n,m.R n m -> R m n
;   Tran:all n,m,k.R n m -> R m k -> R n k
;   n  m  3:R n m

; ?_3: R n m from
;   Symm:all n,m.R n m -> R m n
;   Tran: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 `Tran` 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 `Tran`.

We can now write:

```
(use 3)

> ; ok, ?_3 is proved.  The active goal now is
; ?_4: R m n from
;   Symm:all n,m.R n m -> R m n
;   Tran:all n,m,k.R n m -> R m k -> R n k
;   n  m  3:R n m
>
```

We finally employ `Symm` and another `use`:

```
(use "Symm")
```

```
; ok, ?_4 can be obtained from
; ?_5: R n m from
;   Symm:all n,m.R n m -> R m n
;   Tran: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.
>
```

The same example. We here present a MINLOG proof of the original goal of 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-predconst-name "R" (make-arity (py "nat") (py "nat")))
(set-goal (pf "(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 "Symm")
(name-hyp 3 "Tran")
(assume "n" 4)
(ex-elim 4)
(assume "m" 5)
(cut (pf "R m n"))
(assume 6)
(use-with 3 (pt "n") (pt "m") (pt "n") "?")
(drop "Symm" "Tran" 4)
(split)
(use 5)
(use 6)
(use-with 2 (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 its first occurrence in the proof above we write `"?"` to indicate that MINLOG will have to replace the current goal with a new goal. In the second occurrence of `use-with`, MINLOG will instantiate as specified the universal quantifiers in premise 2 and then use hypothesis 5 to prove the goal.

The command `inst-with` is analogous to `use-with`, but operates for 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.

`ex-elim` eliminates an existential quantifier and produces a new universally quantified goal.

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`. The first 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.

4.2. **Another example with classical logic.** We conclude this section on predicate logic with a final example of a formula which requires classical logic. We want to prove the following:

$$\exists^c x\,.Qx \to \forall y\, Qy,$$

where Q is now a unary predicate, and we do not require Q to range on the natural numbers. In addition, the existential quantifier, $\exists^c$, is here a classical existential quantifier, to be distinguished from the existential quantifier we encountered in the previous example. A classical quantifier $\exists^c x$ is nothing more than an abbreviation for $\neg\,\forall x\,\neg$.

The formula we want to prove is known as the "drinker" formula, as it says something like: "in a bar, there is a person such that if she drinks then everybody drinks".

One could state the goal either by direct use of the classical existential quantifier (called `excl` in MINLOG) or by replacing it with its meaning by use of the universal quantifier. In the first case one would write:

```
(set-goal (pf "excl x. Q x -> all y Q y"))
```

MINLOG would then automatically replace the `excl` quantifier by its meaning when performing the first command.

Otherwise we can state the formula as follows:

```
(set-goal (pf "(all x.(Q x -> all y Q y) -> bot) -> bot"))
```

In our implementation we shall introduce Q as a predicate on an arbitrary type, say α. MINLOG already has a type variable `alpha` as default, and we shall use it in the following (hence no declaration of the type `alpha` is needed).

As the proof is quite simple and does not introduce any new notion, we only write the code, letting the reader verify it in MINLOG.

```
(add-predconst-name "Q" (make-arity (py "alpha")))
(add-var-name "x" "y" (py "alpha"))
(set-goal (pf "(all x.(Q x -> all y Q y) -> bot) -> bot"))
```

```
(assume 1)
(use 1 (pt "x"))
(assume 2 "y")
(use "Stab-Log")
(assume 3)
(use 1 (pt "y"))
(assume 4)
(use "Efq-Log")
(use-with 3 4)
```

We can store the proof as a theorem to be used later on with the command:

```
(save "Drinker")
```

5. Induction

Induction on the natural numbers. We now present a simple proof which exemplifies the use of induction on the natural numbers. The formula to prove is the following:

$$\forall n,\, m.\, n + m = m + n.$$

This can be proved by induction on the natural numbers as follows: we fix an n and show that $n + 0 = 0 + n$ and also that if $n + m = m + n$ then $n + Succ(m) = Succ(m) + n$.

As before we will make use of a file already available in the distribution which contains the definitions of the algebra of the natural numbers and also of the operations of addition and multiplication on the natural numbers. These are defined by means of computation and rewrite rules. If the file is not already loaded[1], we type:

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

We set the goal by letting:

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

We now instantiate the first quantifier with `n`:

```
(assume "n")

; ok, we now have the new goal
; ?_2: all m n+m=m+n from
;   n
>
```

We apply induction by simply typing:

```
(ind)
```

The command `ind` requires a universally quantified goal and applies induction to it in accord to the definition of the specific algebra type (in this case *nat*).

MINLOG's reply will be something like this:

[1]It is a good practice to run a new MINLOG session when loading new files which could turn out to be incompatible with previously loaded files or previous definitions.

```
; ok, ?_2 can be obtained from
; ?_4: all n15.n+n15=n15+n -> n+Succ n15=Succ n15+n from
;   n

; ?_3: n+0=0+n from
;   n
>
```

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

```
(normalize-goal)

; ok, the normalized goal is
; ?_5: T from
;   n
>
```

The latter command can be abbreviated with `ng` and it will normalize the goal by using the computation rules for `+` introduced in the file `nat.scm`.

The goal is now proved by simply appealing to the axiom `Truth-Axiom` (which can be used as an argument to the `use` command).

```
(use "Truth-Axiom")

; ok, ?_5 is proved.  The active goal now is
; ?_4: all n15.n+n15=n15+n -> n+Succ n15=Succ n15+n from
;   n
>
```

We can now instantiate the quantified variable `n15` by `m`:

```
(assume "m" 1)

; ok, we now have the new goal
; ?_6: n+Succ m=Succ m+n from
;   n  m  1:n+m=m+n
>
```

Finally we normalize the goal and use the assumption:

```
(ng)

; ok, the normalized goal is
; ?_7: n+m=m+n from
;   n  m  1:n+m=m+n
>
(use 1)
; ok, ?_7 is proved.  Proof finished.
>
```

To see the proof:

```
(display-proof)
```

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, $2 \cdot n$ is even. We define two new *program constants* (see [**?**]) `Odd` and `Even` which take a natural number as argument and give a boolean (true or false) as output. The behaviour of these program constants can be specified by means of *computation rules*. In this case the computation rules will simultaneously characterize `Odd` and `Even`. The command used to introduce a new program constant is `add-program-constant`. It will require 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 [**?**]). In the following, the type of the new constants `Odd` and `Even` will be introduced by means of the command `mk-arrow`, which produces an arrow type.

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 implementation of the example should clarify how to use these commands.

```
(libload "nat.scm")

(add-program-constant  "Odd" (mk-arrow (py "nat") (py "boole")) 1)
(add-program-constant  "Even" (mk-arrow (py "nat") (py "boole")) 1)
(add-computation-rule (pt "Odd 0") (pt "F"))
(add-computation-rule (pt "Even 0") (pt "T"))
(add-computation-rule (pt "Odd (Succ n)") (pt "Even n"))
(add-computation-rule (pt "Even (Succ n)") (pt "Odd n"))

(set-goal (pf "all n.Even (2*n)"))
(ind)
(ng)
(use "Truth-Axiom")
(ng)
(assume "n" 1)
(use 1)
```

Induction on Lists. The following example is an exercise on lists over an arbitrary type α. Also this example illustrates the use of induction, but since we now deal with *infinitary algebras* (see [**?**]) the task will result a bit harder than when working with the natural numbers.

Together with the file `nat.scm`, we now need to load also the file `list.scm`, which contains basic definitions and operations on lists over an arbitrary type α. We recommend to go through the file before starting to work at this example.

We shall introduce a function, Rev, on lists which has the effect of reverting a list, and then prove the following:

$$\forall y, z.Equal(\mathsf{Rev}\,(y :+: z))((\mathsf{Rev}\,z) :+: (\mathsf{Rev}\,y))),$$

where y and z are lists over an arbitrary type α and :+: denotes the append function on lists as defined in `list.scm`. Note that we here need to use the predicate *Equal* instead of $=$ because the algebra of lists over α is an infinitary algebra and hence equality for it has to be treated as a predicate constant with appropriate axioms (see [**?**]).

Before stating the goal we need to define Rev. This has to be defined by induction, by first giving its value for the empty list and then saying how it applies to a non-empty list. The two defining conditions for Rev will be the following:

$$\mathsf{Rev}\,(\mathsf{Nil}\,\alpha) = (\mathsf{Nil}\,\alpha),$$
$$\mathsf{Rev}\,(a :: y) = (\mathsf{Rev}\,y) :+: (a:)$$

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

First of all we load the files `nat.scm` and `list.scm`.

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

To simplify our work, we declare some variables of the appropriate types:

```
(add-var-name "a" "b" (py "alpha"))
(add-var-name "u" "v" "w" "s" (py "list alpha"))
```

We then declare a new program constant, `Rev`, which takes a list over `alpha` as argument and gives another list over `alpha` as output.

```
(add-program-constant "Rev"
                      (py "list alpha => list alpha") 1 'const 1)
```

We can now introduce two computation rules which correspond to the two conditions for Rev presented above.

```
(add-computation-rule (pt "(Rev alpha) (Nil alpha)")
                      (pt "(Nil alpha)"))
(add-computation-rule (pt "(Rev alpha) (a::w)")
      (pt "((Rev alpha) w) :+: (a:)"))
```

Now we could start proving the goal:

```
all w,s. Equal ((Rev alpha)(w :+: s))
               (((Rev alpha) s) :+: ((Rev alpha) w)))
```

With the purpose of simplifying our proof, we now deliberately introduce three "ad hoc" global assumptions which will be used as lemmata in the proof of the main goal. We prove one of these global assumptions at the end of the main proof, and leave the others as an exercise for the reader.

We can use `aga` as an abbreviation for `add-global-assumption`

```
(aga "Reff" (pf "all a,w,s. (Equal w s)
                            -> Equal (a::w) (a::s)"))

(aga "Eqrev" (pf "all w,s,u. Equal w s
                             -> Equal (w :+: u) (s :+: u)"))
```

```
(aga "Asrev" (pf "all v,s,w,u. Equal v ((s :+: w) :+: u)
                                -> Equal v  (s :+: (w :+: u))"))
```

We now prove a short lemma which says that any list, z, is equal to the list obtained by appending the empty list to z.

```
; Lemma
(set-goal (pf "all s. Equal s (s :+:(Nil alpha))"))
(ind)
(ng)
(use-with "Eq-Refl" (py "list alpha") (pt "(Nil alpha)"))
(assume "a" "w" 1)
(use "Reff")
(use 1)
(save "AppendEmpty")
```

We note that by writing `(use-with "Eq-Refl" (py "list alpha") (pt "(Nil alpha)"))`, we make use of the axiom of reflexivity for `Equal`, which is called `Eq-Refl`, and apply it to the term `Nil alpha` of type `list alpha`.

The lemma and the global assumptions stated above can now be used to prove the goal.

```
(set-goal (pf "all w,s. (Equal ((Rev alpha)(w :+: s))
                        (((Rev alpha) s) :+: ((Rev alpha) w)))"))
(ind)
(ng)
(assume "s")
(use "AppendEmpty")
(assume "a" "w")
(ng)
(assume 1 "s")
(use-with "Asrev" (pt "((Rev alpha)(w :+: s) :+: a:)")
          (pt "((Rev alpha) s)")
          (pt "((Rev alpha) w)") (pt "a:") "?")
(use-with "Eqrev" (pt "((Rev alpha)(w :+: s))")
          (pt "((Rev alpha) s :+: (Rev alpha) w)") (pt "a:") "?")
(use 1)
```

Finally, we show how to prove the global assumption `Eqrev`, as it is a typical application of equality reasoning.

```
(set-goal (pf "all w,s,u. Equal w s
                          -> Equal (w :+: u) (s :+: u)"))
(strip)
(simp 1)
(use-with "Eq-Refl" (py "list alpha") (pt "s:+:u"))
```

Note the command `strip` which moves at one time all universally quantified variables and hypotheses of the current goal into the context.

`simp` simplifies a proof which involves the predicate `Equal`, by substituting 'Equal' terms in the goal.

6. 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. [**?**], [**?**]). The second is given by the command `(search)` and embodies a search algorithm based on Miller's [**?**] and on ideas of Berger (see [**?**] for details on the algorithm and for some differences with Miller's algorithm). The command `search` enables us to automatically find a proof for a wider class of formulas compared with `prop`, since it also works for formulae with quantifiers (see the reference manual for a detailed description of the class of formulae dealt with by `search`).

Prop. One is advised to use `prop` for propositional (minimal) logic. `prop` 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 falsum 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)` after stating the goal or at any point of a proof from which one guesses that (minimal) propositional logic would 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 apply the command to the first examples of propositional logic of this tutorial, including "Peirce's law".

Search. The `search` command is more powerful then `prop` since it allows us to automatically find proofs also for some quantified formulas, but it only operates a search in minimal logic. If one wants to apply this command to a classical formula like "Peirce's law", one could for example add the appropriate instance of "Ex falsum 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 or theorems or even hypothesis from the given context which one would like to be used in the resulting proof.[2] 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.

[2]It is important to notice that in MINLOG we do not quantify over propositions, hence one needs to exercise some care in choosing the global assumptions to be used in `search`. In the case of "Peirce's law", one would need to pass e.g. from the global assumption `Efq-Log` to a theorem which is a specific instance of it, and make use of the latter in `search`.

An example with `search`. As an 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-predconst-name "in" (make-arity (py "alpha") (py "beta")))
(add-var-name "f" (py "alpha=>alpha"))

(set-goal
 (pf
  "all f.(all x,V. in (f x) V -> excl U. in x U
                  & all y. in y U -> in (f y) V) ->
          all x,W. in (f(f x)) W -> excl U. in x U
                  & all y.in y U -> in (f(f y)) W"))
(search)
(dnp)
```

Note that one can switch on a verbose 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.

7. Conclusion

We finally note that one of the main motivations for developing MINLOG and one of its most important features is program extraction, that is, the possibility of using MINLOG to extract functional programs from proof terms. However, the treatment of program extraction would require a consistent extension of this tutorial and it is hence omitted for the time being. The reader willing to know more about this topic is encouraged to consult the reference manual and the relevant papers listed on the MINLOG web page.

To conclude, we recall that all the examples of this tutorial can be found in the file `tutorial.scm` located in the examples directory of the MINLOG distribution.

8. Useful Commands for Emacs and Petite Scheme

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`

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 `Esc` or `Alt`.

9. Useful Commands: 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, we shall write them down, often with a short description of their use gathered from the reference manual. The reader is anyhow advised to check the full details in the reference manual.

10. Some declarations needed to start a proof

`(add-tvar-name` *`name1 ...`*`)`

`(add-alg ...)`, and also
`(add-algs ...) (add-param-alg ...) (add-param-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`)

To each command above there corresponds one to remove constants, variables etc already introduced. For example:

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

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

`(display-program-constants` *`name1 ...`*`)` .

`(display-global-assumptions` *`string1 ...`*`)`

`(display-constructors` *`alg-name1 ...`*`)`

`(display-theorems` *`string1 ...`*`)`

11. Goals

`(set-goal` *formula*`)`
where *formula* needs to be closed (if not universal quantifiers will be inserted automatically).

`(normalize-goal` *goal*`)` (abbr. `ng`)
replaces the goal by its normal form.

`(display-current-goal)` (abbr. `dcg`)

12. 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. One needs to explicitly provide terms for those variables that cannot be automatically instantiated by pattern unification. On the contrary, when pattern

unification succeeds in finding appropriate instances for the quantifiers in the goal, then these instances will be automatically inserted.

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.

13. 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 $A \to B$. Note that the same effect can also be produced by means of the `use` command.

`(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^{boole}, $\neg r^{\mathsf{boole}}$, $t = s$ or $t \approx s$. In case r^{boole}, the boolean term r in the goal is replaced by T, and in case $\neg r^{\mathsf{boole}}$ 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-with x y u)`
is used when proving a goal G from an existential hypothesis $ExHyp$: $\exists yA$. 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)`.

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

14. 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 with the new refined goal, and allows 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.

15. Displaying proofs objects

`(display-proof . opt-proof)` (abbr. `dp`)

`(display-proof-expr . opt-proof)` (abbr. `dpe`)

`(display-eterm . opt-proof)` (abbr. `det`)

`(check-and-display-proof)` (abbr. `cdp`)

`(display-normalized-proof . opt-proof)` (abbr. `dnp`)

`(display-normalized-proof-expr . opt-proof)` (abbr. `dnpe`)

`(display-normalized-eterm . opt-proof)` (abbr. `dnet`)

References

[1] R. Dyckhoff, *Contraction–free sequent calculi for intuitionistic logic*, Journal of Symbolic Logic **57** (1992), no. 3, 795–807.

[2] J. Hudelmaier, *Bounds for cut elimination in intuitionistic propositional logic*, Mathematische Fakultät, Eberhard–Karls–Universität, Tübingen", 1989.

[3] 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.

[4] H. Schwichtenberg, *Minlog reference manual*, LMU München, Mathematisches Institut, Theresienstraße 39, D-81371 München, 2001.